Types: Motivation and Type Checking

Kathi Fisler, modified from CS173 notes by sk and dbtucker

November 12, 2002

1 Types

1.1 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

```
numv-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 a distinction between the *syntactic* and *meta* views of a program. When we implement a language, we are really working with two languages: the one we are implementing, and the one we are writing the implementation in (in this course, Scheme is the latter). The latter is often called the *meta*-language. Whenever we talk about aspects of our interpreter (such as representations, errors, etc), it helps to distinguish which view we are working in. For example, when we implemented assignment using Scheme's boxes, we were using a meta-representation for stores (because the representation came from the implementation language).

In similar fashion, we can ask whether program *errors* are coming from the syntactic or meta levels. The *numv-n* 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}}
```

[1] Not to be confused with a syntax error! (which arises if a program doesn't match the input grammar)

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,

```
{let {f {fun {x} {+ x 1}}}
  {+ 3
     {f 5}}}
```

is clearly legal, whereas

```
{let {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 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
   {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. 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
   {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. 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] 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: $7 \rightarrow 22 \rightarrow 11 \rightarrow 34 \rightarrow 17 \rightarrow 52 \rightarrow 26 \rightarrow 13 \rightarrow 40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$.)

1.2 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

```
{+ 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!

1.3 Typing Rules

Just as we need a language for capturing the semantics of a language, we need a language to capture the typing restrictions in a language. 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. 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,

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

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

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

[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.

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:

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

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).

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 (→), where the type before the arrow represents the argument and that after the arrow represents the function's return value:

type ::= number
  | (type → type)

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

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

in $(\mathsf{number} \rightarrow (\mathsf{number} \rightarrow \mathsf{number}))$, while the inner function has type $(\mathsf{number} \rightarrow \mathsf{number})$. 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:

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

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.

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. 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.

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?

2 Implementing Typing Rules: A Simple Type Checker

Typing rules look similar to the judgements we wrote to describe language semantics. We can implement semantics judgements in the form of an interpreter. Similarly, we can implement type rules (or type judgements) via a program called a type-checker. Actually, there are many different ways to turn typing rules into programs that perform type checking; we'll look at different approaches in this segment of the course. For sake of concreteness, however, let's consider one for now.

The previous section added type annotations to our syntax for function declarations. If we want to write a type checker, we must extend our datatype for functions to capture those annotations:

```
(define-datatype TFunExp TFunExp?
  [numE (n number?)]
  [varE (v symbol?)]
  [addE (lhs TFunExp?)
        (rhs TFunExp?)]
  [funE (param symbol?)
        (paramtype type?)
        (returntype type?)
        {body TFunExp?}]
  [appE (fun TFunExp?)
        (arg TFunExp?)])
```

Note that the only case that changed from our old *AFunExp* definition is the *funE* case, as that is the only case to which we've added annotations. Note also that the new fields are of a new datatype called *type*, so we have to define that as well.

```
(define-datatype type type?
   [numberT]
   [functionT (from type?)
              (to type?)])
```

Given these definitions, a basic type checker could walk the datastructure for a program checking that each kind of expression respects its typing rules. Here's the skeleton of that program that handles the *numE* and *addE* cases.

```
;; type-check : TFunExp → type
;; returns type of given expression or raises error if expression
;; violates the typing rules
(define (type-check texp)
  (cases TFunExp texp
    [numE (n) (numberT)]
    [varE (v) . . . ]
    [addE (lhs rhs)
      (cond [(and (numberT? (type-check lhs))
                  (numberT? (type-check rhs)))
             (numberT)]
            [else (error 'type-check "+ received non-numeric argument")])]
    [funE (param paramtype returntype body) . . . ]
    [appE (fun arg) . . . ]))
```

It may seem strange that our type checker returns a type rather than a boolean (indicating whether the program checked correctly). The latter approach would require us to annotate *every* expression with its type (which would get tedious and cumbersome!). Whenever we leave some expressions un-annotated (such as +), our type checker is actually doing a combination of type inference based on what we know of the primitives and type checking of user-defined functions and variables. The contract on *type-check* reflects this combination.

3 Influences on Type System Design

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).

5 Typing the Infinite Loop

Given the language *TFunExp* (typed *AFunExp*), can we write a recursive program? Let's just try to write an infinite loop. Our first attempt might be this *AFunExp* program

```
  {let {f {fun {i}
             {f i}}}
    {f 10}}
```

which, expanded out, becomes

```
{{fun {f}
   {f 10}}
 {fun {i}
   {f i}}}
```

When we place type annotations on this program, we get

```
{{fun {f : (number -> number)} : number
   {f 10}}
 {fun {i : number} : number
   {f i}}}
```

These last two steps don't matter, of course. This program doesn't result in an infinite loop, because the `f` in the body of the function isn't bound, so after the first iteration, the program halts with an error.

As an aside, this error is easier to see in the typed program: when the type checker tries to check the type of the annotated program, it finds no type for `f` on the last line. Therefore, it would halt with a type error, preventing this erroneous program from ever executing.[4]

Okay, that didn't work, but we knew about that problem: we saw it before when introducing recursion. At the time, we asked you to consider whether it was possible to write a recursive function without an explicit recursion construct. We have since seen that this is possible from the lambda calculus lecture (using the *Y combinator*[5]). Without going into the full glory of Y, we can write an infinite loop by exploiting its central idea, namely self-application:

```
  {let {omega {fun {x}
                 {x x}}}
    {omega omega}}
```

How does this work? Simply substituting `omega` with the function, we get

[4] In this particular case, however, a simpler check would prevent the erroneous program from starting to execute, namely checking to ensure there are no free variables. Since the identifier is free in the last line, the program will halt with an unbound identifier error if execution ever reaches that code fragment.

[5] A *combinator* is simply a procedure that has no free variables. Why would it have such a name?

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

Substituting again, we get

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

and so on. In other words, this program executes forever! It is conventional to call the function ω (lower-case omega), and the entire expression Ω (upper-case omega).[6]

Okay, so Ω seems to be our ticket. This is clearly an infinite loop in *AFunExp*. All we need to do is convert it to *TFunExp*, which is simply a matter of annotating all procedures. Since there's only one, ω, this should be especially easy.

To annotate ω, we must provide a type for the argument and one for the result. Let's call the argument type, namely the type of `x`, τ_a and that of the result τ_r. The body of ω is `{x x}`. From this, we can conclude that τ_a must be a function (arrow) type, since we use `x` in the function position of an application. That is, τ_a has the form $\tau_1 \rightarrow \tau_2$, for some τ_1 and τ_2 yet to be determined.

What can we say about τ_1 and τ_2? τ_1 must be whatever type `x`'s argument has. Since `x`'s argument is itself `x`, τ_1 must be the same as the type of `x`. We just said that `x` has type τ_a. This immediately implies that

$$\tau_a = \tau_1 \rightarrow \tau_2 = \tau_a \rightarrow \tau_2$$

In other words,

$$\tau_a = \tau_a \rightarrow \tau_2$$

What type can we write that satisfies this equation? In fact, no types in our type language can satisify it, because this type is recursive without a base case. Any type we try to write will end up being infinitely long. Since we cannot write an infinitely long type (recall that we're trying to annotate ω, so if the type is infinitely long, we'd never get around to finishing the text of the program!), it follows by contradiction[7] that ω and Ω cannot be typed in our type system, and therefore their corresponding programs are not programs in *TFunExp*. (We are being rather lax here—what we've provided is informal reasoning, not a proof—but such a proof does exist.)

6 Termination

We concluded our exploration of the type of Ω by saying that the annotation on the argument of ω must be infinitely long. A curious reader ought to ask, is there any connection between the boundlessness of the type and the fact that we're trying to perform a non-terminating computation? Or is it mere coincidence? In fact, it's not—*TFunExp* is a rather strange language!

TFunExp, which is a first cousin of a language you'll sometimes see referred to as the "simply-typed lambda calculus",[8] enjoys a rather interesting property: it is said to be *strongly normalizing*. This intimidating term says of a programming language that no matter what program you write in the language, it will *always terminate*!

To understand why this property holds, think about our type language. The only way to create compound types is through the function constructor. But every time we apply a function, we discharge one function constructor: that is, we "erase an arrow". Therefore, after a finite number of function invocations, the computation must "run out of arrows".[9] Because only function applications can keep a computation running, the computation is forced to terminate.

This is a *very* informal argument for why this property holds—it is cetainly far from a proof (though, again, formal proofs of this property do exist). However, it does help us see why we must inevitably have bumped into an infinitely long type while trying to annotate the infinite loop. This formal property also tells us that, no matter how valid our reasoning about the particulars of typing Ω, in general, it is impossible. We will need a different language if we must write infinite loops.

[6] Strictly speaking, it's anachronistic to refer to the lower and upper "case" for the Greek alphabet, since it predates moveable type in the West by two millennia. I'm not sure why Greek has two cases, anyway.

[7] We implicitly assumed it would be possible to annotate ω and explored what that type annotation would be. The contradiction is that no such annotation is possible.

[8] Why "simply"? You'll see what other options there are later.

[9] Oddly, this seems to never happen to the heroes of Indian and other ancient mythologies whose warriors fought with bows.

What good is a language without infinite loops?!? Well, think how often you encountered a program that truly does (or, rather, is intended to) run *forever* . . . That's what we thought. (You should actually be able to come up with plenty of answers: operating systems, for instance!) In contrast, there are lots of programs that we would like to ensure will *not* run forever. These include:

- real-time systems
- program linkers
- packet filters in network stacks
- client-side Web scripts
- network routers
- photocopier (and other) device initialization
- configuration files (such as Makefiles)

and so on. That's what makes the simply-typed lambda calculus so neat: instead of pondering and testing endlessly (no pun intended), we get mathematical certitude that, with a correct implementation of the type checker, no infinite loops can sneak past us.

Of course, programming in this language can be a bit unwieldy. Nevertheless, it may still be useful to think of it as an "object code", namely as a target for compilers of higher-level languages. Many people now use C, which was intended for humans, as a mere back-end for their compilers (where C takes the place of assembly language, being more portable and offering slightly better features such as register allocation). Likewise, in many domains, it may make sense to offer a better surface syntax, but use the simply-typed lambda calculus as a back-end target for a compiler. Indeed, next week we will see how effective a translator of this form can be, up to the point of inferring the type annotations!

Puzzle

- We've been told that the Halting Problem is undecidable. Yet here we have a language accompanied by a theorem that proves that all programs will terminate. In particular, then, the Halting Problem is not only very decidable, it's actually quite simple: In response to the question "Does this program halt", the answer is *always* (a loud and affirmative) "Yes!" What gives? Did the folks who wrote down the Halting Problem miss something? (Think this through carefully. You should be able to offer a very precise answer.)
- While the simply-typed lambda calculus is fun to discuss, it may not be the most pliant programming language, even as the target of a compiler (much less something programmers write explicitly). Partly this is because it doesn't quite focus on the right problem. To a Web browsing user, for instance, what matters is whether a downloaded program runs *immediately*. If it doesn't, then sometimes whether it runs after five minutes or five days, or even loops forever, scarcely matterss, as the user's attention has flitted to some other topic.

  Consequently, a better variant of the lambda calculus might be one whose types reflect *resources*, such as time and space. The "type" checker would then ask the user running the program for resource bounds, then determine whether the program can actually execute within the provided resources. Can you design and implement such a language? Can you write useful programs in it?

7 Typed Recursive Programming

The last time we introduced recursion as an explicit language construct, it was because our limited imaginations did not figure out Y on their own. To compensate, we added a recursion construct. We later discovered that Y would have accomplished the same effect (conceptually, even if not in terms of efficiency, say). This time, however, there is no lurking, secret way of reclaiming recursion: strong normalization says we really are sunk. So we'll have to add recursion again, explicitly, this time because we really don't have a choice.

To do this, we'll simply reintroduce our `rec` construct to define the language *TFunRecExp*. The BNF for the language is

```
<TFunRecExp> ::= <num>
           | {+ <TFunRecExp> <TFunRecExp>}
           | {fun {<id> : <type>} : <type> <TFunRecExp>}
           | {<TFunRecExp> <TFunRecExp>}
           | {rec {<id>  : <type> <TFunRecExp>} <TFunRecExp>}
```

where

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

Note that the `rec` construct now needs an explicit type annotation also. As an example program that uses `rec`, consider the following program that sums all numbers smaller than a given number (the name Sigma mimics the mathematical Σ used for this purpose):

```
{rec {Sigma : {number -> number}
              {fun {n : number} : number {+ n {Sigma {+ n -1}}}}}
  {Sigma 5}}
```

Two observations arise with this example: first, we are now able to write (and type) programs that will not terminate; second, some of our type information may appear redundant between the `rec` and `fun` expressions, but it is not redundant in the general case (can you construct an example?).

What is the type judgment for `rec`? It must be of the form

$$\frac{???}{\Gamma\vdash\{rec\ \{i : \tau_i\ v\}\ b\} : \tau}$$

since we want to conclude something about the entire term. What goes in the antecedent? We can determine this more easily by realizing that a `rec` is a bit like an immediate function application—after all, it's a refinement of `with`. So just as with functions, we're going to have *assumptions* and *guarantees*, just both in the same rule.

We want to assume that τ_i is a legal annotation, and use that to check the body; but we also want to guarantee that τ_i is a legal annotation. Let's do them in that order. The former is relatively easy:

$$\frac{\Gamma[i\leftarrow\tau_i]\vdash b : \tau \qquad ???}{\Gamma\vdash\{rec\ \{i : \tau_i\ v\}\ b\} : \tau}$$

Now let's hazard a guess about the form of the latter:

$$\frac{\Gamma[i\leftarrow\tau_i]\vdash b : \tau \qquad \Gamma\vdash v : \tau}{\Gamma\vdash\{rec\ \{i : \tau_i\ v\}\ b\} : \tau}$$

But what the structure of the term named by v? Surely it has references to the identifier named by i in it, but i is almost certainly not bound in Γ (and even if it is, it's not bound to the value we want for i). Therefore, we'll have to extend Γ with a binding for i—not surprising, if you think about the scope of i in a `rec` term—to check v also:

$$\frac{\Gamma[i\leftarrow\tau_i]\vdash b : \tau \qquad \Gamma[i\leftarrow\tau_i]\vdash v : \tau}{\Gamma\vdash\{rec\ \{i : \tau_i\ v\}\ b\} : \tau}$$

Is that right? Do we want v to have type τ, the type of the entire expression? Not quite: we want it to have the type we promised it would have, namely τ_i:

$$\frac{\Gamma[i\leftarrow\tau_i]\vdash b : \tau \qquad \Gamma[i\leftarrow\tau_i]\vdash v : \tau_i}{\Gamma\vdash\{rec\ \{i : \tau_i\ v\}\ b\} : \tau}$$

Now we can understand how the typing of recursion works. We extend the environment not once, but twice. The extension to type b is the one that *initiates* the recursion; the extension to type v is the one that *sustains* it. Both extensions are therefore necessary. And because a type checker doesn't actually run the program, it doesn't need an infinite number of arrows. When type checking is done and execution begins, the run-time system does, in some sense, need "an infinite quiver of arrows", but we've already seen how to implement that using two different representations of environments!

Exercises

Typing recursion looks deceptively simple, but it's actually worth studying in a bit of detail. Take a simple example such as Ω and work through the rules:

- Does the expression named by v have to be a procedure? Do the typing rules for `rec` depend on this? And if not, can you write a judgment for `rec` that forces v to syntactically be a procedure?
- First, write Ω with type annotations so it passes the type checker.
- Second, trace through the type rules to make sure you understand why this version of Ω types. Draw the *tree* of judgments applied and discharged. This tree, like botanical ones but unlike other ones in computer science, actually grows upward; its leaves are *axioms*, or rules with no antecedents (so they are always true). The axioms in our type system are the judgments for identifiers and numbers.
- Define the BNF entry and generate a type judgment for `with` in the typed language.

8 Recursive Types

8.1 Declaring Recursive Types

We saw in the previous section how `rec` was necessary to write recursive *programs*. But what about defining recursive *types*? Recursive types are present all over computer science: even basic data structures like lists and trees are recursive (since the rest of a list is also a list, and each sub-tree is itself a tree).

Suppose we try to type the program

```
{rec {sum : ???
       {fun {L : ???} : number
         {if {empty? L}
             0
             {+ {first L} {sum {rest L}}}}}}
  {sum {Cons 1 {Cons 2 {Cons 3 Empty}}}}}
```

(We've taken generous liberties in this program, assuming the existence of ordinary conditionals and list primitives.)

First, consider how our existing typing rules will work on this program: what will the rule for `+` require? Looking back to the rules (or our *type-check* program that implemented them), `{first L}` will have to be a number in order for this program to satisfy the rules. This means that we really have a list of numbers. Let's rewrite the program to make this expectation explicit: the primitives are defined only for lists of numbers.

```
{rec {sum : ???
       {fun {L : ???} : number
         {if {numempty? L}
             0
             {+ {numfirst L} {sum {numrest L}}}}}}
  {sum {numCons 1 {numCons 2 {numCons 3 numEmpty}}}}}
```

(Ignore concerns about how to generalize lists to contain other types – we'll come back to that issue next week.)

Now, what should we write in place of the question marks? Let's consider the type of `L`. What kind of value can be an argument to `L`? Clearly a numeric cons, because that's the argument supplied in the first invocation of `sum`. But eventually, a numeric empty is passed to `L` also. This means `L` needs to have *two* types: (numeric) cons and empty.

In languages like ML (and Java), procedures do not consume arguments of more than one distinct type. Instead, they force programmers to define a new type that encompasses all the possible arguments. This is precisely what a datatype definition, of the kind we have been writing in Scheme, permits us to do. So let's try to write down such a datatype in a hypothetical extension to our (typed) implemented language:

```
{datatype numList
    {[numEmpty]
```

```
     [numCons {fst : number}
              {rst : ???}]}
  {rec {sum : (numList -> number) ...}
    {sum ...}}}
```

We assume that a datatype declaration introduces a collection of *variants*, followed by an actual body that uses the datatype. What type annotation should we place on `rst`? This should be precisely the new type we are introducing, namely `numList`.

A datatype declaration therefore enables us to do a few distinct things all in one notation:

1. Give names to new types.
2. Introduce conditionally-defined types (*variants*).
3. Permit recursive definitions.

If these are truly distinct, we should consider whether there are more primitive operators that we may provide, so a programmer may assemble a datatype if that's what they need, but they could possibly use the primitives to assemble other types also.

But how distinct are these three operations, really? Giving a type a new name would be only so useful if the type were simple (for instance, creating the name `bool` as an alias for `boolean` may be convenient, but it's certainly not conceptually earth-shattering), so this capability is most useful when the name is assigned to a complex type. Recursion needs a name to use for declaring self-references, so it depends on the ability to introduce a new name. Finally, well-founded recursion depends on having both recursive and non-recursive cases, meaning the recursive type must be defined as a collection of variants (of which at least one is not self-referential). So the three capabilities coalesce very nicely.

8.2 Judgments for Recursive Types

Let's consider another example of a recursive type: a family tree.

```
{datatype FamilyTree
    {[unknown]
     [person {name : string}
             {mother : FamilyTree}
             {father : FamilyTree}]}
  ...}
```

This data definition allows us to describe as much of the genealogy as we know, and terminate the construction when we reach an unknown person. What type declarations ensue from this definition?

$$unknown : \rightarrow FamilyTree$$

$$person : string \times FamilyTree \times FamilyTree \rightarrow FamilyTree$$

This doesn't yet give us a way of distinguishing between the two variants, and of selecting the fields in each variant. In Scheme, we use **cases** to perform both of these operations. A corresponding case dispatcher for the above datatype might look like

```
{FamilyTree-cases v
  [{unknown} ...]
  [{person n m f} ...]}
```

Its pieces would be typed as follows:

$$\frac{\Gamma \vdash v : FamilyTree \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma[n \leftarrow \mathsf{string}, m \leftarrow FamilyTree, f \leftarrow FamilyTree] \vdash e_2 : \tau}{\Gamma \vdash \{\textit{FamilyTree-cases}\ v\ \{[unknown]\ e_1\}\ \{[person\ n\ m\ f]\ e_2\}\} : \tau}$$

In other words, to type the entire `cases` statement to type τ, we first ensure that the value being dispatched is of the right type. Then we must make sure each branch of the switch returns a τ.[10] We can ensure that by checking each of the bodies in the right type environment. Because `unknown` has no fields, its `cases` branch binds no variables, so we check e_1 in Γ. In the branch for `person`, however, we bind three variables, so we must check the type of e_2 in a suitably extended Γ.

Though the judgment above is for a very specific type declaration, the general principle should be clear from what we've written. Effectively, the type checker introduces a new type rule for each typed `cases` statement based on the type declaration at the time it sees the declaration. Writing the judgment above in terms of subscripted parameters is tedious but easy.

Given the type rules above, consider the following program:

```
{datatype FamilyTree
    {[unknown]
     [person {name : string}
             {mother : FamilyTree}
             {father : FamilyTree}]}
  {person "Mitochondrial Eve" {unknown} {unknown}}}
```

What is the type of the expression in the body of the datatype declaration? It's $FamilyTree$. But when the value escapes from the body of the declaration, how can we access it any longer? (We assume that the type checker renames types consistently, so $FamilyTree$ in one scope is different from $FamilyTree$ in another scope—just because the names are the same, the types should not conflated automatically.) It basically becomes an *opaque type* that is no longer usable. This is not very useful at all![11]

At any rate, the type checker permitted a program that is quite useless, and we might want to prevent this. Therefore, we could place the restriction that the type defined in the datatype (in this case, $FamilyTree$) should be different from the type of the expression body τ. This prevents programmers from inadvertently returning values that nobody else can use.

Obviously, this restriction doesn't reach far enough. Returning a vector of $FamilyTree$ values avoids the restriction above, but the effect is the same: no part of the program outside the scope of the datatype can use these values. So we may want a more stringent restriction: the type being different *should not appear free* in τ.

This restriction may be overreaching, however. For instance, a programmer might define a new type, and return a package (a vector, say) consisting of two values: an instance of the new type, and a procedure that accesses the instances. For instance,

```
{datatype FamilyTree
    {[unknown]
     [person {name : string}
             {mother : FamilyTree}
             {father : FamilyTree}]}
  {let {unknown-person : FamilyTree {unknown}}
    {vector
      {person "Mitochondrial Eve"
              unknown-person
              unknown-person}
      {fun {v : FamilyTree} : string
        {FamilyTree-cases v
          [{unknown}      {error ...}]
          [{person n m f} n]}}}}}}
```

In this vector, the first value is an instance of $FamilyTree$, while the second value is a procedure of type

$$FamilyTree \rightarrow \mathsf{string}$$

[10] Based on the preceding discussion, if the two cases needed to return different types of values, how would you address this need in a language that enforced the type judgment above?

[11] Actually ... that's not quite true. You could use this to define the essence of a module or object system. These are called *existential types*. But we won't study them further in this course.

Other values, such as `unknown-person`, are safely hidden from access. If we lift the restriction of the previous paragraph, this becomes a legal pair of values to return from an expression. Notice that the pair in effect form an *object*: you can't look into it, the only way to access it is with the "public" procedure. Indeed, this kind of type definition sees use in defining object systems.

That said, we still don't have a clear description of what restriction to affix on the type judgment for datatypes. Modern programming languages address this quandary by affixing no restriction at all. Instead, they force all type declarations to be at the "top" level. Consequently, no type name is ever unbound, so the issues of this section do not arise.

8.3 Space for Datatype Variant Tags

One of the benefits programmers incur from using datatypes—beyond the error checking—is slightly better space consumption. (Note: "better space consumption" = "using less space".) Without datatypes, memory management systems need tags that indicate both the type *and* the variant (such as 'num-empty and 'num-cons versus just *empty* and *cons*), we now need to store *only* the variant. Why? Because the type checker statically ensures that we won't pass the wrong kind of value to procedures! Therefore, the run-time system needs to use only as many bits as are necessary to distinguish between *all the variants of a type*, rather than all datatypes. Since the number of variants is usually quite small, of the order of 3-4, the number of bits necessary for the tags is usually small also.

We are now taking a big risk, however. In the liberal tagging regime, where we use both type and variant tags, we can be sure a program will never execute on the wrong kind of data. But if we switch to a more conservative tagging regime—where we don't store type tags also—we run a huge risk. If we perform an operation on a value of the wrong type, we may completely destroy our data. For instance, suppose we can somehow pass a $NumList$ to a procedure expecting a $FamilyTree$. If the `FamilyTree-cases` operation looks only at the variant bits, it could end up accessing a `numCons` as if it were a `person`. But a `numCons` has only two fields; when the program accesses the third field of this variant, it is essentially getting junk values. Therefore, we have to be very careful performing these kinds of optimizations. How can we be sure they are safe?

9 Formalizing the Type Abstraction

We would like some kind of guarantee that the program will fail to exhibit certain errors (or, to put it more positively, that it will execute *without* certain errors, such as inadvertently accessing a $NumList$ as if it were a $FamilyTree$). In particular, we would like to know that the type system did indeed abstract over values: that running the type checker *correctly predicted* (up to the limits of the abstraction) what the program would do. We call this property of a type system *type soundness*[12]:

> For all programs p, if the type of p is τ, then p will evaluate to a value that has type τ.

Note that the statement of type soundness *connects types with execution*. This tells the user that the type system is not some airy abstraction: what it predicts has bearing on "practice", namely program execution.

We have to be somewhat careful about how we define type soundness. For instance, we say above (emphasis added) "p *will* evaluate to a value such that …". But what if the program doesn't terminate? So we must recast this statement to say

> For all programs p, if the type of p is τ and p evaluates to v, then $v : \tau$.[13]

Actually, this isn't quite true either. What if the program executes an expression like (*first empty*)? There are a few options open to the language designer:

- Return a value such as -1. We hope you cringe at this idea! It means a program that fails to check return values everywhere will produce nonsensical results.

[12] The term "soundness" comes from logic, where it has a very specific meaning. Probably related is the British usage, found in writings of authors like P.G. Wodehouse, of referring to a person as "a sound egg".

[13] We could write this more explicitly as: 'For all programs p, if the type checker assigns p the type τ, and the semantics say that p evaluates to a value v, then the type checker will also assign v the type τ."

- Diverge, i.e., go into an infinite loop. This approach is used by theoreticians (study the statement of type soundness carefully and you can see why), but as software engineers we should soundly reject this.

- Raise an exception. This is the preferred modern solution.

Raising exceptions means the program does not terminate with a value, nor does it not terminate. So we have to refine this statement still further:

> For all programs p, if the type of p is τ, p will, if it terminates, either evaluate to a value v such that $v : \tau$, or raise one of a well-defined set of exceptions.

The exceptions are a bit of a cop-out, because we can move arbitrarily many errors into that space. In Scheme, for instance, the trivial type checker rejects no programs, and all errors fall under the exceptions. In contrast, researchers work on very sophisticated languages where some traditional exceptions (such as violating array bounds) become type errors. This last phrase of the type soundness statement gives lots of room for type system design.

As software engineers, we should care deeply about type soundness. To paraphrase Robin Milner, who first proved a modern language's soundness (specifically, ML),

> Well-typed programs do not go wrong.

This is partially rhetorical, but there is an underlying truth here: a program that passes the type checker (and is thus "well-typed") absolutely cannot exhibit certain classes of mistakes. (The choice of the word "wrong" is definitely an exaggeration, since it suggests that types ensure "correctness". But we should not let this verbal quibble overshadow Milner's achievement.)

Why is type soundness not obvious? Consider the following simple program (the details of the numbers aren't relevant):

```
{if0 {+ 1 2}
     {{fun {x : number} : number {+ 1 x}} 3}
     {{fun {x : number} : number {+ 1 {+ 2 x}}} 1}}
```

During execution, the program will explore only one branch of the conditional:

$$\dfrac{\dfrac{\texttt{1},\emptyset \Rightarrow \texttt{1} \qquad \texttt{2},\emptyset \Rightarrow \texttt{2}}{\texttt{\{+ 1 2\}},\emptyset \Rightarrow \texttt{3}} \qquad \dfrac{\ldots}{\texttt{\{\{fun ...\} 1\}},\emptyset \Rightarrow \texttt{4}}}{\texttt{\{if0 \{+ 1 2\} \{\{fun ...\} 3\} \{\{fun ...\} 1\}\}},\emptyset \Rightarrow \texttt{4}}$$

but the type checker must explore both:

$$\dfrac{\dfrac{\emptyset \vdash \texttt{1} : \textsf{number} \qquad \emptyset \vdash \texttt{2} : \textsf{number}}{\emptyset \vdash \texttt{\{+ 1 2\}} : \textsf{number}} \qquad \dfrac{\ldots}{\emptyset \vdash \texttt{\{\{fun ...\} 3\}} : \textsf{number}} \qquad \dfrac{\ldots}{\emptyset \vdash \texttt{\{\{fun ...\} 1\}} : \textsf{number}}}{\emptyset \vdash \texttt{\{if0 \{+ 1 2\} \{\{fun ...\} 3\} \{\{fun ...\} 1\}\}} : \textsf{number}}$$

Furthermore, even for each expression, the proof trees in the semantics and the type world will be quite different. As a result, it is *far from obvious* that the two systems will have any relationship in their answers. This is why a theorem is not only necessary, but sometimes also difficult to prove.

Type soundness is, then, really a claim that the type system and run-time system (as represented by the semantics) are in sync. The type system erects certain abstractions, and the theorem states that the run-time system mirrors those abstractions. Most modern languages, like ML and Java, have this flavor.

In contrast, C and C++ *do not have sound type systems*. That is, the type system may define certain abstractions, but the run-time system does not honor and protect these. (In C++ it sort of does for object types, but not for types inherited from C.) This is a particularly insidious kind of language, because the static type system lulls the programmer into thinking it will detect certain kinds of errors, but fails to deliver on that promise during execution.

Actually, the reality of C is much more complex: C has *two different type systems*. There is one type system (with types such as `int`, `double` and even function types) at the level of the program, and a different type system, defined *solely* by lengths of bitstrings, at the level of execution. This is a kind of "bait-and-switch" operation on the part of the language. As a result, it isn't even meaningful to talk about soundness for C, because the static types and dynamic type representations simply don't agree. Instead, the C run-time system simply interprets bit sequences according to specified static types. (Procedures like `printf` are notorious at this: if you ask to print using the specifier `%s`, `printf` will simply print a sequence of characters until it hits a null-teriminator: never mind that the value you were pointing to was actually a double! This is of course why C is very powerful at low-level programming tasks, but how often do you actually need such power?)

To summarize all this, we introduce the notion of *type safety*:

> Type *safety* is the property that no primitive operation is ever applied to values of the wrong type.

By primitive operation we mean not only addition and so forth, but also procedure application. In short, *a safe language honors the abstraction boundaries it erects*. Since abstractions are crucial for designing and maintaining large systems, safety is a key software engineering attribute in a language. (Even most C++ libraries are safe, but the problem is you have to be sure nobody in some legacy C library isn't performing unsafe operations.) Using this concept, we can construct the following table:

	statically checked	not statically checked
type safe	ML, Java	Scheme
type unsafe	C, C++	assembly

The important thing to remember is, due to the Halting Problem, some checks simply can never be performed statically; something must always be deferred to execution time. The trade-off in type design is to minimize the time and space consumed by these objects during execution (and, for that matter, how many guarantees a type system can tractably give a user)—in particular, in shuffling where the set of checked operations lies between static and dynamic checking.

So what is "strong typing"? As best as we can tell, this is a meaningless phrase, and people often use it in a nonsensical fashion. To some it seems to mean "The language has a type checker". To others it means "The language is sound" (that is, the type checker and run-time system are related). To most, it seems to just mean, "A language like Pascal, C or Java, related in a way I can't quite make precise". For amusement at cocktail parties, when someone mentions the phrase "strongly typed", ask them to define it and catch the errors and inconsistencies. And please, don't use the term yourself unless you want to sound poorly-trained and ignorant. Use the terminology of this course instead.