PL: Lecture #26

Tuesday, November 26th

Typing Data

> ☞ ***PLAI §27***

An important concept that we have avoided so far is user-defined types. This issue exists in practically all languages, including the ones we did so far, since a language without the ability to create new user-defined types is a language with a major problem. (As a side note, we did talk about mimicking an object system using plain closures, but it turns out that this is insufficient as a replacement for true user-defined types — you can kind of see that in the Schlac language, where the lack of all types mean that there is no type error.)

In the context of a statically typed language, this issue is even more important. Specifically, we talked about typing recursive code, but we should also consider typing recursive data. For example, we will start with a `length` function in an extension of the language that has `empty?`, `rest`, and `NumCons` and `NumEmpty` constructors:

```
{rec {length : ???
      {fun {l : ???} : Number
        {if {empty? l}
          0
          {+ 1 {call length {rest l}}}}}}
  {call length {NumCons 1 {NumCons 2 {NumCons 3 {NumEmpty}}}}}}
```

But adding all of these new functions as built-ins is getting messy: we want our language to have a form for defining new kinds of data. In this example — we want to be able to define the `NumList` type for lists of numbers. We therefore extend the language with a new `with-type` form for creating new user-defined types, using variants in a similar way to our own course language:

```
{with-type {NumList [NumEmpty]
                    [NumCons Number ???]}
  {rec {length : ???
        {fun {l : ???} : Number
          ...}}
    ...}}
```

We assume here that the `NumList` definition provides us with a number of new built-ins — `NumEmpty` and `NumCons` constructors, and assume also a `cases` form that can be used to both test a value and access its components (with the constructors serving as patterns). This makes the code a little different than what we started with:

```
{with-type {NumList [NumEmpty]
                    [NumCons Number ???]}
  {rec {length : ???
        {fun {l : ???} : Number
          {cases l
            [{NumEmpty}    0]
            [{NumCons x r} {+ 1 {call length r}}]}}}
    {call length {NumCons 1 {NumCons 2 {NumCons 3 {NumEmpty}}}}}}}
```

The question is what should the `???` be filled with? Clearly, recursive data types are very common and we need to support them. The scope of `with-type` should therefore be similar to `rec`, except that it works at the type level: the new type is available for its own definition. This is the complete code now:

```
{with-type {NumList [NumEmpty]
                    [NumCons Number NumList]}
  {rec {length : (NumList -> Number)
        {fun {l : NumList} : Number
          {cases l
            [{NumEmpty}    0]
            [{NumCons x r} {+ 1 {call length r}}]}}}
    {call length {NumCons 1 {NumCons 2 {NumCons 3 {NumEmpty}}}}}}}
```

(Note that in the course language we can do just that, and in addition, the `Rec` type constructor can be used to make up recursive types.)

An important property that we would like this type to have is for it to be *well founded*: that we'd never get stuck in some kind of type-level infinite loop. To see that this holds in this example, note that some of the variants are self-referential (only `NumCons` here), but there is at least one that is not (`NumEmpty`) — if there wasn't any simple variant, then we would have no way to construct instances of this type to begin with!

[As a side note, if the language has lazy semantics, we could use such types — for example:

```
{with-type {NumList [NumCons Number NumList]}
  {rec {ones : NumList {NumCons 1 ones}}
    ...}}
```

Reasoning about such programs requires more than just induction though.]

Judgments for recursive types

If we want to have a language that is basically similar to the course language, then — as seen above — we'd use a similar `cases` expression. How should we type-check such expressions? In this case, we want to verify this:

```
Γ ⊢ {cases l [{NumEmpty} 0]
             [{NumCons x r} {+ 1 {call length r}}]} : Number
```

Similarly to the judgment for `if` expressions, we require that the two result expressions are numbers. Indeed, you can think about `cases` as a more primitive tool that has the functionality of `if` — in other words, given such user-defined types we could implement booleans as a new type and and implement `if` using `cases`. For example, wrap programs with:

```
{with-type {Bool [True] [False]} ...}
```

and translate `{if E1 E2 E3}` to `{cases E1 [{True} E2] [{False} E3]}`.

Continuing with typing `cases`, we now have:

```
Γ ⊢ 0 : Number            Γ ⊢ {+ 1 {call length r}} : Number
————————————————————————————————————————————————————————————————
Γ ⊢ {cases l [{NumEmpty} 0]
             [{NumCons x r} {+ 1 {call length r}}]} : Number
```

But this will not work — we have no type for `r` here, so we can't prove the second subgoal. We need to consider the `NumList` type definition as something that, in addition to the new built-ins, provides us with type judgments for these built-ins. In the case of the `NumCons` variant, we know that using `{NumCons x r}` is a pattern that matches `NumList` values that are a result of this variant constructor but it also binds `x` and `r` to the values of the two fields, and since all uses of the constructor are verified, the fields have the declared types. This means that we need to extend Γ in this rule so we're able to prove the two subgoals. Note that we do the same for the `NumEmpty` case, except that there are no new bindings there.

```
Γ ⊢ 0 : Number
Γ[x:=Number; r:=NumList] ⊢ {+ 1 {call length r}} : Number
——————————————————————————————————————————————————————————
Γ ⊢ {cases l [{NumEmpty} 0]
             [{NumCons x r} {+ 1 {call length r}}]} : Number
```

Finally, we need to verify that the value itself — `l` — has the right type: that it is a `NumList`.

```
Γ ⊢ l : NumList
Γ ⊢ 0 : Number
Γ[x:=Number; r:=NumList] ⊢ {+ 1 {call length r}} : Number
——————————————————————————————————————————————————————————
Γ ⊢ {cases l [{NumEmpty} 0]
             [{NumCons x r} {+ 1 {call length r}}]} : Number
```

But why `NumList` and not some other defined type? This judgment needs to do a little more work: it should inspect all of the variants that are used in the branches, find the type that defines them, then use that type as the subgoal. Furthermore, to make the type checker more useful, it can check that we have complete coverage of the variants, and that no variant is used twice:

```
Γ ⊢ l : NumList
    (also need to show that NumEmpty and NumCons are all of
    the variants of NumList, with no repetition or extras.)
Γ ⊢ 0 : Number
Γ[x:=Number; r:=NumList] ⊢ {+ 1 {call length r}} : Number
——————————————————————————————————————————————————————————
Γ ⊢ {cases l [{NumEmpty} 0]
             [{NumCons x r} {+ 1 {call length r}}]} : Number
```

Note that how this is different from the version in the textbook — it has a `type-case` expression with the type name mentioned explicitly — for example: `{type-case l NumList {{NumEmpty} 0} ...}`. This is essentially the same as having each defined type come with its own `cases` expression. Our rule needs to do a little more work, but overall it is a little easier to use. (And the same goes for the actual implementation of the two languages.)

In addition to `cases`, we should also have typing judgments for the constructors. These are much simpler, for example:

```
Γ ⊢ x : Number   Γ ⊢ r : NumList
————————————————————————————————
  Γ ⊢ {NumCons x r} : NumList
```

Alternatively, we could add the constructors as new functions instead of new special forms — so in the Picky language they'd be used in `call` expressions. The `with-type` will then create the bindings for its scope at runtime, and for the typechecker it will add the relevant types to Γ:

```
Γ[NumCons:=(Number NumList -> NumList); NumEmpty:=(-> NumList)]
```

(This requires functions of any arity, of course.) Using accessor functions could be similarly simpler than `cases`, but less convenient for users.

Note about representation: a by-product of our type checker is that whenever we have a `NumList` value, we know that it *must* be an instance of either `NumEmpty` or `NumCons`. Therefore, we could represent such values as a wrapped value container, with a single bit that distinguishes the two. This is in contrast to dynamically typed languages like Racket, where every new type needs to have its own globally unique tag.

"Runaway" instances extra

Consider this code:

```
{with-type {NumList [NumEmpty] ...} {NumEmpty}}
```

We now know how to type check its validity, but what about the type of this whole expression? The obvious choice would be `NumList`:

```
{with-type {NumList [NumEmpty] ...} {NumEmpty}} : NumList
```

There is a subtle but important problem here: the expression evaluates to a `NumList`, but we can no longer use this value, since we're out of the scope of the `NumList` type definition! In other words, we would typecheck a program that is pretty much useless.

Even if we were to allow such a value to flow to a different context with a `NumList` type definition, we wouldn't want the two to be confused — following the principle of lexical scope, we'd want each type definition to be unique to its own scope even if it has the same concrete name. For example, using `NumList` as the type of the inner `with-type` here:

```
{with-type {NumList something-completely-different}
  {with-type {NumList [NumEmpty] ...}
    {NumEmpty}}}
```

would make it wrong.

(In fact, we might want to have a new type even if the value goes outside of this scope and back in. The default struct definitions in Racket have exactly this property — they're *generative* — which means that each "call" to `define-struct` creates a new type, so:

```
(define (two-foos)
  (define (foo x)
    (struct foo (x))
    (foo x))
  (list (foo 1) (foo 2)))
```

returns two instances of two *different* `foo` types!)

One way to resolve this is to just forbid the type from escaping the scope of its definition — so we would forbid the type of the expression from being `NumList`, which makes

```
{with-type {NumList [NumEmpty] ...} {NumEmpty}} : NumList
```

invalid. But that's not enough — what about returning a compound value that *contains* an instance of `NumList`? For example — what if we return a list or a function with a `NumList` instance?

```
{with-type {NumList [NumEmpty] ...}
  {fun {x} {NumEmpty}}} : Num -> NumList??
```

Obviously, we would need to extend this restriction: the resulting type should not mention the defined type *at all* — not even in lists or functions or anything else. This is actually easy to do: if the overall expression is type-checked in the surrounding lexical scope, then it is type-checked in the surrounding type environment (Γ), and that environment has nothing in it about `NumList` (well, nothing about *this* `NumList`).

Note that this is, very roughly speaking, what our course language does: `define-type` can only define new types when it is used at the top-level.

This works fine with the above assumption that such a value would be completely useless — but there are aspects of such values that are useful. Such types are close to things that are known as "existential types", and they are for defining opaque values that you can do nothing with except pass them around, and only code in a specific lexical context can actually use them. For example, you could lump together the value with a function that can work on this value. If it wasn't for the `define-type` top-level restriction, we could write the following:

```
(: foo : Integer -> (List ??? (??? -> Integer)))
(define (foo x)
  (define-type FOO [Foo Integer])
```

```
    (list (Foo 1)
          (lambda (f)
            (cases f [(Foo n) (* n n)]))))
```

There is nothing that we can do with resulting `Foo` instance (we don't even have a way to name it) — but in the result of the above function we get also a function that could work on such values, even ones from different calls:

```
  ((second (foo 1)) (first (foo 2))) -> 4
```

Since such kind of values are related to hiding information, they're useful (among other things) when talking about module systems (and object systems), where you want to have a local scope for a piece of code with bindings that are not available outside it.

Type soundness

> ☞ ***PLAI §28***

Having a type checker is obviously very useful — but to be able to *rely* on it, we need to provide some kind of a formal account of the kind of guarantees that we get by using one. Specifically, we want to guarantee that a program that type-checks is guaranteed to *never* fail with a type error. Such type errors in Racket result in an exception — but in C they can result in anything. In our simple Picky implementation, we still need to check the resulting value in `run`:

```
  (typecheck prog (NumT) (EmptyTypeEnv))
  (let ([result (eval prog (EmptyEnv))])
    (cases result
      [(NumV n) n]
      ;; this error is never reached, since we make sure
      ;; that the program always evaluates to a number above
      [else (error 'run "evaluation returned a non-number: ~s"
                           result)]))
```

A soundness proof for this would show that checking the result (in `cases`) is not needed. However, the check must be there since Typed Racket (or any other typechecker) is far from making up and verifying such a proof by itsef.

In this context we have a specific meaning for "fail with a type error", but these failures can be very different based on the kind of properties that your type checker verifies. This property of a type system is called *soundness*: a *sound* type system is one that will never allow such errors for type-checked code:

> *For any program `p`, if we can type-check `p : τ`, then `p` will evaluate to a value that is in the type `τ`.*

The importance of this can be seen in that it is the *only* connection between the type system and code execution. Without it, a type system is a bunch of syntactic rules that are completely disconnected from how the program runs. (Note also that — "in the type" — works for the (common) case where types are sets of values.)

But this statement isn't exactly what we need — it states a property that is too strong: what if execution gets stuck in an infinite loop? (That wasn't needed before we introduced `rec`, where we could extend the conclusion part to: "… then `p` will terminate and evaluate to a value that is in the type `τ`".) We therefore need to revise it:

> *For any program `p`, if we can type-check `p : τ`, and if `p` terminates and returns `v`, then `v` is in the type `τ`.*

But there are still problems with this. Some programs evaluate to a value, some get stuck in an infinite loop, and some … throw an error. Even with type checking, there are still cases when we get runtime errors. For example, in practically all statically typed languages the length of a list is not encoded in its type, so `{first null}` would throw an error. (It's possible to encode more information like that in types, but there is a downside to this too: putting more information in the type system means that things get less flexible, and it becomes more difficult to write programs since you're moving towards proving more facts about them.)

Even if we were to encode list lengths in the type, we would still have runtime errors: opening a missing file, writing to a read-only file fetching a non-existent URL, etc, so we must find some way to account for these errors. Some "solutions" are:

- For all cases where an error should be raised, just return some value (of the appropriate type). For example, `(first l)` could return `0` if the list is empty; `(substring "foo" 10 20)` would return "huh?", etc. It seems like a dangerous way to resolve the issue, but in fact that's what most C library calls do: return some bogus value (for example, `malloc()` returns `NULL` when there is no available memory), and possibly set some global flag that specifies the exact error. (The main problem with this is that C programmers often don't check all of these conditions, leading to propagating undetected errors further down — and all of this is a very rich source of security issues.)
- For all cases where an error should be raised, just get stuck into an infinite loop. This approach is obviously impractical — but it is actually popular in some theoretical circles. The reason for that is that theory people will often talk about "domains", and to express facts about computation on these domains, they're extended with a "bottom" value that represents a diverging computation. Since this introduction is costly in terms of work that it requires, adding one more such value can lead to more effort than re-using the same "bottom" value.
- Raise an exception. This works out better than the above two extremes, and it is the approach taken by practically all modern languages.

So, assuming exceptions, we need to further refine what it means for a type system to be sound:

> *For any program `p`, if we can type-check `p : τ`, and if `p` terminates without exceptions and returns `v`, then `v` is in the type `τ`.*

An important thing to note here is that languages can have very different ideas about where to raise an exception. For example, Scheme implementations often have a trivial type-checker and throw runtime exceptions when there is a type error. On the other hand, there are systems that express much more in their type system, leaving much less room for runtime exceptions.

A soundness proof ties together a particular type system with the statement that it is sound. As such, it is where you tie the knot between type checking (which happens at the syntactic level) and execution (dealing with runtime values). These are two things that are usually separate — we've seen throughout the course many examples for things that could be done only at runtime, and things that should happen completely on the syntax. `eval` is the important *semantic function* that connects the two worlds (`compile` also did this, when we converted our evaluator to a compiler) — and in here, it is the soundness proof that makes the connection.

To demonstrate the kind of differences between the two sides, consider an `if` expression — when it is executed, only one branch is evaluated, and the other is irrelevant, but when we check its type, *both* sides need to be verified. The same goes for a function whose execution get stuck in an infinite loop: the type checker will not get into a loop since it is not executing the code, only scans the (finite) syntax.

The bottom line here is that type soundness is really a claim that the type system provides some guarantees about the runtime behavior of programs, and its proof demonstrates that these guarantees do hold. A fundamental problem with the type system of C and C++ is that it is not sound: these languages *have* a type system, but it does not provide such runtime guarantees. (In fact, C is even worse in that it really has two type systems: there is the system that C programmers usually interact with, which has a conventional set of type — including even higher-order function types; and there is the machine-level type system, which only talks about various bit lengths of data. For example, using "%s" in a printf() format string will blindly copy characters from the address pointed to by the argument until it reaches a 0 character — even if the actual argument is really a floating point number or a function.)

Note that people often talk about "strongly typed languages". This term is often meaningless in that different people take it to mean different things: it is sometimes used for a language that "has a static type checker", or a language that "has a non-trivial type checker", and sometimes it means that a language has a sound type system. For most people, however, it means some vague idea like "a language like C or Pascal or Java" rather than some concrete definition.

Explicit polymorphism extra

> ☞ ***PLAI §29***

Consider the `length` definition that we had — it is specific for `NumList`s, so rename it to `lengthNum`:

```
{with-type {NumList ...}
  {rec {lengthNum : (NumList -> Num)
        {fun {l : NumList} : Num
          {cases l
            [{NumEmpty}    0]
            [{NumCons x r} {+ 1 {call lengthNum r}}]}}}
    {call lengthNum
          {NumCons 1 {NumCons 2 {NumCons 3 {NumEmpty}}}}}}}
```

To simplify things, assume that types are previously defined, and that we have an even more Racket-like language where we simply write a `define` form:

```
{define lengthNum
  {fun {l : NumList} : Num
    {cases l
      [{NumEmpty}    0]
      [{NumCons x r} {+ 1 {call lengthNum r}}]}}}
```

What would happen if, for example, we want to take the length of a list of booleans? We won't be able to use the above code since we'd get a type error. Instead, we'd need a separate definition for the other kind of length:

```
{define lengthBool
  {fun {l : BoolList} : Num
    {cases l
      [{BoolEmpty}    0]
      [{BoolCons x r} {+ 1 {call lengthBool r}}]}}}
```

We've designed a statically typed language that is effective in catching a large number of errors, but it turns out that it's too restrictive — we cannot implement a single generic `length` function. Given that our type system allows an infinite number of types, this is a major problem, since every new type that we'll want to use in a list requires writing a new definition for a length function that is specific to this type.

One way to address the problem would be to somehow add a new `length` primitive function, with specific type rules to make it apply to all possible types. (Note that the same holds for the list type too — we need a new type definition for each of these, so this solution implies a new primitive type that will do the same generic trick.) This is obviously a bad idea: there are other functions that will need the same treatment (`append`, `reverse`, `map`, `fold`, etc), and there are other types with similar problems (any new container type). A good language should allow writing such a length function inside the language, rather than changing the language for every new addition.

Going back to the code, a good question to ask is what is it exactly that is different between the two `length` functions? The answer is that there's very little that is different. To see this, we can take the code and replace all occurrences of `Num` or `Bool` by some `???`. Even better — this is actually abstracting over the type, so we can use a familiar type variable, τ:

```
{define length⟨τ⟩
  {fun {l : ⟨τ⟩List} : Num
    {cases l
      [{⟨τ⟩Empty}    0]
      [{⟨τ⟩Cons x r} {+ 1 {call length⟨τ⟩ r}}]}}}
```

This is a kind of a very low-level "abstraction" — we replace parts of the text — parts of identifiers — with a kind of a syntactic meta variable. But the nature of this abstraction is something that should look familiar — it's abstracting over the code, so it's similar to a macro. It's not really a macro in the usual sense — making it a real macro involves answering questions like what does `length` evaluate to (in the macro system that we've seen, a macro is not something that is a value in itself), and how can we use these macros in the `cases` patterns. But still, the similarity should provide a good intuition about what goes on — and in particular the basic fact is the same: this *is* an abstraction that happens at the syntax level, since typechecking is something that happens at that level.

To make things more manageable, we'll want to avoid the abstraction over parts of identifiers, so we'll move all of the meta type variables, and make them into arguments, using `⟨...⟩` brackets to stand for "meta level applications":

```
{define length⟨τ⟩
  {fun {l : List⟨τ⟩} : Num
    {cases l
      [{Empty⟨τ⟩}    0]
      [{Cons⟨τ⟩ x r} {+ 1 {call length⟨τ⟩ r}}]}}}
```

Now, the first "⟨τ⟩" is actually a kind of an input to `length`, it's a binding that has the other `τ`s in its scope. So we need to have the syntax reflect this somehow — and since `fun` is the way that we write such abstractions, it seems like a good choice:

```
{define length
  {fun {τ}
    {fun {l : List⟨τ⟩} : Num
      {cases l
        [{Empty⟨τ⟩}    0]
        [{Cons⟨τ⟩ x r} {+ 1 {call length⟨τ⟩ r}}]}}}}
```

But this is very confused and completely broken. The new abstraction is not something that is implemented as a function — otherwise we'll need to somehow represent type values within our type system. (Trying that has some deep problems — for example, if we have such type values, then it needs to have a type too; and if we add some `Type` for this, then `Type` itself should be a value — one that has *itself* as its type!)

So instead of `fun`, we need a new kind of a syntactic, type-level abstraction. This is something that is acts as a function that gets used by the type checker. The common way to write such functions is with a capital `lambda` — `Λ`. Since we already use Greek letters for things that are related to types, we'll use that as is (again, with "⟨⟩"s), instead of a confusing capitalized `Lambda` (or a similarly confusing `Fun`):

```
{define length
  ⟨Λ ⟨τ⟩                   ; sidenote: similar to (All (t) ...)
    {fun {l : List⟨τ⟩} : Num
      {cases l
        [{Empty⟨τ⟩}    0]
        [{Cons⟨τ⟩ x r} {+ 1 {call length⟨τ⟩ r}}]}}⟩}
```

and to use this `length` we'll need to instantiate it with a specific type:

```
{+ {call length⟨Num⟩ {list 1 2}}
   {call length⟨Bool⟩ {list #t #f}}}
```

Note that we have several kinds of meta-applications, with slightly different intentions:

- length⟨τ⟩ is the recursive call, which needs to keep using the same type that initiated the `length` call. It makes sense to have it there, since `length` is itself a type abstraction.
- List⟨τ⟩ is using `List` as if it's also this kind of an abstraction, except that instead of abstracting over some generic code, it abstracts over a generic type. This makes sense too: it naturally leads to a

generic definition of `List` that works for all types since it is also an abstraction.

- Finally there are `Empty⟨τ⟩` and `Cons⟨τ⟩` that are used for patterns. This might not be necessary, since they are expected to be variants of the `List⟨τ⟩` type. But if we were doing this without pattern matching (for example, see the book) then we'd need `null?` and `rest` functions. In that case, the meta application would make sense — `null?⟨τ⟩` and `rest⟨τ⟩` are the τ-specific versions of these functions which we get with this meta-application, in the same way that using `length` needs an explicit type.

Actually, the last item points at one way in which the above sample calls:

```
{+ {call length⟨Num⟩ {list 1 2}}
   {call length⟨Bool⟩ {list #t #f}}}
```

are broken — we should also have a type argument for `list`:

```
{+ {call length⟨Num⟩ {list⟨Num⟩ 1 2}}
   {call length⟨Bool⟩ {list⟨Bool⟩ #t #f}}}
```

or, given that we're in the limited picky language:

```
{+ {call length⟨Num⟩ {cons⟨Num⟩ 1 {cons⟨Num⟩ 2 null⟨Num⟩}}}
   {call length⟨Bool⟩ {cons⟨Bool⟩ #t {cons⟨Bool⟩ #f null⟨Bool⟩}}}}
```

Such a language is called "parametrically polymorphic with explicit type parameters" — it's *polymorphic* since it applies to any type, and it's *explicit* since we have to specify the types in all places.

Polymorphism in the type description language extra

Given our definition for `length`, the type of `length⟨Num⟩` is obvious:

```
length⟨Num⟩ : List⟨Num⟩ -> Num
```

but what would be the type of `length` by itself? If it was a function (which was a broken idea we've seen), then we would write:

```
length : τ -> (List⟨τ⟩ -> Num)
```

But this is broken in the same way: the first arrow is fundamentally different than the second — one is used for a `Λ`, and the other for a `fun`. In fact, the arrows are even more different, because the two `τ`s are very different: the first one *binds* the second. So the first arrow is bogus — instead of an arrow we need some way to say that this is a type that "for all τ" is "List⟨τ⟩ -> Num". The common way to write this should be very familiar:

```
length : ∀τ. List⟨τ⟩ -> Num
```

Finally, `τ` is usually used as a meta type variable; for these types the convention is to use the first few Greek letters, so we get:

```
length : ∀α. List⟨α⟩ -> Num
```

And some more examples:

```
filter : ∀α. (α->Bool) × List⟨α⟩ -> List⟨α⟩
map : ∀α,β. (α->β) × List⟨α⟩ -> List⟨β⟩
```

where × stands for multiple arguments (which isn't mentioned explicitly in Typed Racket).

Type judgments for explicit polymorphism and execution

extra

Given our notation for polymorphic functions, it looks like we're introducing a runtime overhead. For example, our `length` definition:

```
{define length
  ⟨Λ ⟨α⟩
    {fun {l : List⟨α⟩} : Num
      {cases l
        [{Empty⟨α⟩}    0]
        [{Cons⟨α⟩ x r} {+ 1 {call length⟨α⟩ r}}]}}⟩}
```

looks like it now requires another curried call for each iteration through the list. This would be bad for two reasons: first, one of the main goals of static type checking is to *avoid* runtime work, so adding work is highly undesirable. An even bigger problem is that types are fundamentally a syntactic thing — they should not exist at runtime, so we don't want to perform these type applications at runtime simply because we don't want types to exist at runtime. If you think about it, then every traditional compiler that typechecks code does so while compiling, not when the resulting compiled program runs. (A recent exception in various languages are "dynamic" types that are used in a way that is similar to plain (untyped) Racket.)

This means that we want to eliminate these applications in the typechecker. Even better: instead of complicating the typechecker, we can begin by applying all of the type meta-applications, and get a result that does not have any such applications or any type variables left — then use the simple typechecker on the result. This process is called "type elaboration".

As usual, there are two new formal rules for dealing with these abstractions — one for type abstractions and another for type applications. Starting from the latter:

```
   Γ ⊢ E : ∀α.τ
———————————————————
Γ ⊢ E⟨τ₂⟩ : τ[τ₂/α]
```

which means that when we encounter a type application $E\langle\tau_2\rangle$ where E has a polymorphic type $\forall\alpha.\tau$, then we substitute the type variable α with the input type τ_2. Note that this means that conceptually, the typechecker is creating all of the different (monomorphic) `length` versions, but we don't need all of them for execution — having checked the types, we can have a single `length` function which would be similar to the function that Racket uses (i.e., the same "low level" code with types erased).

To see how this works, consider our length use, which has a type of `∀α. List⟨α⟩ -> Num`. We get the following proof that ends in the exact type of `length` (remember that when you prove you climb up):

```
Γ ⊢ length : ∀α. List⟨α⟩ -> Num
——————————————————————————————————————————————
Γ ⊢ length⟨Bool⟩ : (List⟨α⟩ -> Num)[Bool/α]
——————————————————————————————————————————————
Γ ⊢ length⟨Bool⟩ : List⟨Bool⟩ -> Num     [...]
——————————————————————————————————————————————
Γ ⊢ {call length⟨Bool⟩ {cons⟨Bool⟩ ...}} : Num
```

The second rule for type abstractions is:

```
   Γ[α] ⊢ E : τ
———————————————————
Γ ⊢ ⟨Λ⟨α⟩ E⟩ : ∀α.τ
```

This rule means that to typecheck a type abstraction, we need to check the body while binding the type variable α — but it's not bound to some specific type. Instead, it's left unspecified (or non-deterministic) — and typechecking is expected to succeed without requiring an actual type. If some specific type is actually required, then typechecking should fail. The intuition behind this is that a polymorphic function can be one only if it doesn't need some specific type — for example, `{fun {x} {- {+ x 1} 1}}` is an identity function, but it's an identity that requires the input to be a number, and therefore it cannot have a polymorphic ∀α.α type like `{fun {x} x}`.

Another example is our `length` function — the actual type that the list holds better not matter, or our `length` function is not really polymorphic. This makes sense: to typecheck the function, this rule means that we need to typecheck the body, with α being some unknown type that cannot be used.

One thing that we need to be careful when applying any kind of abstraction (and the first rule does just that for a very simple lambda-calculus-like language) is infinite loops. But in the case of our type language, it turns out that this lambda-calculus that gets used at the meta-level is one of the strongly normalizing kinds, therefore no infinite loops happen. Intuitively, this means that we should be able to do this elaboration in just one pass over the code. Furthermore, there are no side-effects, therefore we can safely cache the results of applying type abstraction to speed things up. In the case of `length`, using it on a list of `Num` will lead to one such application, but when we later get to the recursive call we can reuse the (cached) first result.

Explicit polymorphism conclusions extra

Quoted directly from the book:

> *Explicit polymorphism seems extremely unwieldy: why would anyone want to program with it? There are two possible reasons. The first is that it's the only mechanism that the language designer gives for introducing parameterized types, which aid in code reuse. The second is that the language includes some additional machinery so you don't have to write all the types every time. In fact, C++ introduces a little of both (though much more of the former), so programmers are, in effect, manually programming with explicit polymorphism virtually every time they use the STL (Standard Template Library). Similarly, the Java 1.5 and C# languages support explicit polymorphism. But we can possibly also do better than foist this notational overhead on the programmer.*