PL: Lecture #24

Tuesday, November 26th

Types

> ☞ ***PLAI §24***

In our Toy language implementation, there are certain situations that are not covered. For example,

```
{< {+ 1 2} 3}
```

is not a problem, but

```
{+ {< 1 2} 3}
```

will eventually use Racket’s addition function on a boolean value, which will crash our evaluator. Assuming that we go back to the simple language we once had, where there were no booleans, we can still run into errors — except now these are the errors that our code raises:

```
{+ {fun {} 1} 2}
```

or

```
{1 2 3}
```

or

```
{{fun {x y} {+ x y}} 5}
```

In any case, it would be good to avoid such errors right from the start — it seems like we should be able to identify such bad code and not even try to run it. One thing that we can do is do a little more work at parse time, and declare the `{1 2 3}` program fragment as invalid. We can even try to forbid

```
{bind {{x 1}} {x 2 3}}
```

in the same way, but what should we do with this? —

```
{fun {x} {x 2 3}}
```

The validity of this depends on how it is used. The same goes for some invalid expressions — the above bogus expression can be fine if it’s in a context that shadows `<`:

```
{bind {{< *}}
  {+ {< 1 2} 3}}
```

Finally, consider this:

```
{+ 3 {if <mystery> 5 {fun {x} x}}}
```

where mystery contains something like `random` or `read`. In general, knowing whether a piece of code will run with no errors is a problem that is equivalent to the halting problem — and because of this, there is no way to create an “exact” type system: they are all either too restrictive (rejecting programs that would run with no errors) or too permissive (accepting programs that might crash). This is a very practical issue — type safety means a lot less bugs in the system. A good type system is still an actively researched problem.

What is a Type?

> ☞ ***PLAI §25***

A type is any property of a program (or an expression) that can be determined without running the program. (This is different than what is considered a `type` in Racket which is a property that is known only at run-time, which means that before run-time we know nothing so in essence we have a single type (in the static sense).) Specifically, we want to use types in a way that predicts some aspects of the program's behavior, for example, whether a program will crash.

Usually, types are being used as the kind of value that an expression can evaluate to, not the precise value itself. For example, we might have two kinds of values — functions and numbers, and we know that addition always operates on numbers, therefore

```
{+ 1 {fun {x} x}}
```

is a type error. Note that to determine this we don't care about the actual function, just the fact that it is a function.

Important: types can discriminate certain programs as invalid, but they cannot discriminate correct programs from incorrect ones. For example, there is no way for any type system to know that this:

```
{fun {x} {+ x 1}}
```

is an incorrect decrease-by-one function.

In general, type systems try to get to the optimal point where as much information as possible is known, yet the language is not too restricted, no significant computing resources are wasted, and programmers don't spend much time annotating their code.

Why would you want to use a type system?

- Catch errors even in code that you don't execute, for example, when your tests are too weak (but they do *not* substitute proper test suites).
- They help reduce the time spent on debugging (when they detect legitimate errors, rather than force you to change your code).
- As we have seen, they help in documenting code (but they do *not* substitute proper documentation).
- Compilers can use type information to make programs run faster.
- They encourage a more organized code development process. For example, our use of `define-type` and `cases` (inspired by ML) help guide your code. (But note that the actual code can be as disorganized as usual, typechecking or not…)

Our Types — The Picky Language

The first thing we need to do is to agree on what types are. Earlier, we talked about two types: numbers and functions (ignore booleans or anything else for now), we will use these two types for now.

> *In general, this means that we are using the Types are Sets meaning for types, and specifically, we will be implmenting a type system known as a Hindley-Milner system. This is not what Typed Racket is using. In fact, one of the main differences is that in our type system each binding has exactly one type, whereas in Typed Racket an identifier can have different types in different places in the code. An example of this is something that we've talked about earlier:*

```
(: foo : (U String Number) -> Number)
(define (foo x)          ; \ these `x`s have a
  (if (number? x)        ; / (U Number String) type
    (+ x 1)              ; > this one is a Number
    (string-length x)))  ; > and this one is a String
```

A type system is presented as a collection of rules called “type judgments”, which describe how to determine the type of an expression. Beside the types and the judgments, a type system specification needs a (decidable) algorithm that can assign types to expressions.

Such a specification should have one rule for every kind of syntactic construct, so when we get a program we can determine the precise type of any expression. Also, these judgments are usually recursive since a type judgment will almost always rely on the types of sub-expressions (if any).

For our restricted system, we have two rules that we can easily specify:

```
n : Number  (any numeral `n' is a number)
{fun {x} E} : Function
```

(These rules are actually “axioms”, since the state facts that are true by themselves, with no need for any further work.)

And what about an identifier? Well, it is clear that we need to keep some form of an environment that will keep an account of types assigned to identifiers (note: all of this is not at run-time). This environment is used in all type judgments, and usually written as a capital Greek Gamma character (in some places `G` is used to stick to ASCII texts).

The conventional way to write the above two axioms is:

```
Γ ⊢ n : Number
Γ ⊢ {fun {x} E} : Function
```

The first one is read as “Gamma proves that `n` has the type `Number`”. Note that this is a syntactic environment, much like DE-ENVs that you have seen in homework.

So, we can write a rule for identifiers that simply has the type assigned by the environment:

```
Γ ⊢ x : Γ(x)    ; "Γ(x)" is "lookup(x, Γ)" for a new "type env"
```

We now need a rule for addition and a rule for application (note: we’re using a very limited subset of our old language, where arithmetic operators are not function applications). Addition is easy: if we can prove that both `a` and `b` are numbers in some environment Γ, then we know that `{+ a b}` is a number in the same environment. We write this as follows:

```
Γ ⊢ A : Number   Γ ⊢ B : Number
———————————————————————————————
     Γ ⊢ {+ A B} : Number
```

Now, what about application? We need to refer to some arbitrary type now, and the common letter for that is a Greek lowercase tau:

```
Γ ⊢ F : Function   Γ ⊢ V : τᵥ
—————————————————————————————
    Γ ⊢ {call F V} : ???
```

that is — if we can prove that `f` is a function, and that `v` is a value of some type τ_a, then … ??? Well, we need to know more about `f`: we need to know what type it consumes and what type it returns. So a simple `function` is not enough — we need some sort of a function type that specifies both input and output types. We will use the notation that was seen throughout the semester and dump `function`. Now we can write:

```
Γ ⊢ F : (τ₁ -> τ₂)  Γ ⊢ V : τ₁
——————————————————————————————
```

```
     Γ ⊢ {call F V} : τ₂
```

which makes sense — if you take a function of type `τ₁->τ₂` and you feed it what it expects, you get the obvious output type. But going back to the language — where do we get these new arrow types from? We will modify the language and require that every function specifies its input and output type (and assume we have only one argument functions). For example, we will write something like this for a function that is the curried version of addition:

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

So: the revised syntax for the limited language that contains only additions, applications and single-argument functions, and for fun — go back to using the `call` keyword is. The syntax we get is:

```
<PICKY> ::= <num>
          | <id>
          | { + <PICKY> <PICKY> }
          | { fun { <id> : <TYPE> } : <TYPE> <PICKY> }
          | { call <PICKY> <PICKY> }

<TYPE>  ::= Number
          | ( <TYPE> -> <TYPE> )
```

and the typing rules are:

```
Γ ⊢ n : Number

Γ ⊢ {fun {x : τ₁} : τ₂ E} : (τ₁ -> τ₂)

Γ ⊢ x : Γ(x)

Γ ⊢ A : Number   Γ ⊢ B : Number
———————————————————————————————
     Γ ⊢ {+ A B} : Number

Γ ⊢ F : (τ₁ -> τ₂)  Γ ⊢ V : τ₁
——————————————————————————————
     Γ ⊢ {call F V} : τ₂
```

But we're still missing a big part — the current axiomatic rule for a `fun` expression is too weak. If we use it, we conclude that these expressions:

```
{fun {x : Number} : (Number -> Number)
  3}
{fun {x : Number} : Number
  {call x 2}}
```

are valid, as well concluding that this program:

```
{call {call {fun {x : Number} : (Number -> Number)
              3}
            5}
      7}
```

is valid, and should return a number. What's missing? We need to check that the body part of the function is correct, so the rule for typing a `fun` is no longer a simple axiom but rather a type judgment. Here is how we check the body instead of blindly believing program annotations:

```
          Γ[x:=τ₁] ⊢ E : τ₂            ; Γ[x:=τ₁] is
———————————————————————————————————————  ;     extend(Γ, x, τ₁)
Γ ⊢ {fun {x : τ₁} : τ₂ E} : (τ₁ -> τ₂)  ; for the new type envs
```

That is — we want to make sure that if `x` has type `τ₁`, then the body expression `E` has type `τ₂`, and if we can prove this, then we can trust these annotations.

There is an important relationship between this rule and the `call` rule for application:

- In this rule we assume that the input will have the right type and guarantee (via a proof) that the output will have the right type.
- In the application rule, we guarantee (by a proof) an input of the right type and assume a result of the right type.

(Side note: Racket comes with a contract system that can identify type errors dynamically, and assign blame to either the caller or the callee — and these correspond to these two sides.)

Note that, as we said, `number` is really just a property of a certain kind of values, we don't know exactly what numbers are actually used. In the same way, the arrow function types don't tell us exactly what function it is, for example, `(Number -> Number)` can indicate a function that adds three to its argument, subtracts seven, or multiplies it by 7619. But it certainly contains much more than the previous naive `function` type. (Consider also Typed Racket here: it goes much further in expressing facts about code.)

For reference, here is the complete BNF and typing rules:

```
<PICKY> ::= <num>
          | <id>
          | { + <PICKY> <PICKY> }
          | { fun { <id> : <TYPE> } : <TYPE> <PICKY> }
          | { call <PICKY> <PICKY> }

<TYPE>  ::= Number
          | ( <TYPE> -> <TYPE> )

Γ ⊢ n : Number

Γ ⊢ x : Γ(x)

Γ ⊢ A : Number   Γ ⊢ B : Number
———————————————————————————————
     Γ ⊢ {+ A B} : Number

          Γ[x:=τ₁] ⊢ E : τ₂
———————————————————————————————————————
Γ ⊢ {fun {x : τ₁} : τ₂ E} : (τ₁ -> τ₂)

Γ ⊢ F : (τ₁ -> τ₂)  Γ ⊢ V : τ₁
——————————————————————————————
     Γ ⊢ {call F V} : τ₂
```

Examples of using types (abbreviate `Number` as `Num`) — first, a simple example:

```
                {} ⊢ 5 : Num   {} ⊢ 7 : Num
                ———————————————————————————
{} ⊢ 2 : Num         {} ⊢ {+ 5 7} : Num
———————————————————————————————————————————
          {} ⊢ {+ 2 {+ 5 7}} : Num
```

and a little more involved one:

```
     [x:=Num] ⊢ x : Num   [x:=Num] ⊢ 3 : Num
     ———————————————————————————————————————
            [x:=Num] ⊢ {+ x 3} : Num
———————————————————————————————————————————————
{} ⊢ {fun {x : Num} : Num {+ x 3}} : Num -> Num   {} ⊢ 5 : Num
——————————————————————————————————————————————————————————————
      {} ⊢ {call {fun {x : Num} : Num {+ x 3}} 5} : Num
```

Finally, try a buggy program like

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

and see where it is impossible to continue.

The main thing here is that to know that this is a type error, we have to prove that there is no judgment for a certain type (in this case, no way to prove that a `fun` expression has a `Num` type), which we (humans) can only do by inspecting all of the rules. Because of this, we need to also add an algorithm to our type system, one that we can follow and determine when it gives up.

Typing control

☞ ***PLAI §26***

We will now extend our typed Picky language to have a conditional expression, and predicates. First, we extend the BNF with a predicate expression, and we also need a type for the results:

```
<PICKY> ::= <num>
          | <id>
          | { + <PICKY> <PICKY> }
          | { < <PICKY> <PICKY> }
          | { fun { <id> : <TYPE> } : <TYPE> <PICKY> }
          | { call <PICKY> <PICKY> }
          | { if <PICKY> <PICKY> <PICKY> }

<TYPE>  ::= Number
          | Boolean
          | ( <TYPE> -> <TYPE> )
```

Initially, we use the same rules, and add the obvious type for the predicate:

```
Γ ⊢ A : Number   Γ ⊢ B : Number
———————————————————————————————
     Γ ⊢ {< A B} : Boolean
```

And what should the rule for `if` look like? Well, to make sure that the condition is a boolean, it should be something of this form:

```
Γ ⊢ C : Boolean   Γ ⊢ T : ???   Γ ⊢ E : ???
———————————————————————————————————————————
           Γ ⊢ {if C T E} : ???
```

What would be the types of `t` and `e`? A natural choice would be to let the programmer use any two types:

```
Γ ⊢ C : Boolean   Γ ⊢ T : τ₁   Γ ⊢ E : τ₂
—————————————————————————————————————————
          Γ ⊢ {if C T E} : ???
```

But what would the return type be? This is still a problem. (BTW, some kind of a union would be nice, but it has some strong implications that we will not discuss.) In addition, we will have a problem detecting possible errors like:

```
{+ 2 {if <mystery> 3 {fun {x} x}}}
```

Since we know nothing about the condition, we can just as well be conservative and force both arms to have the same type. The rule is therefore:

```
Γ ⊢ C : Boolean   Γ ⊢ T : τ   Γ ⊢ E : τ
———————————————————————————————————————
          Γ ⊢ {if C T E} : τ
```

— using the same letter indicates that we expect the types to be identical, unlike the previous attempt. Consequentially, this type system is fundamentally weaker than Typed Racket which we use in this class.

Here is the complete language specification with this extension:

```
<PICKY> ::= <num>
          | <id>
          | { + <PICKY> <PICKY> }
          | { < <PICKY> <PICKY> }
          | { fun { <id> : <TYPE> } : <TYPE> <PICKY> }
          | { call <PICKY> <PICKY> }
          | { if <PICKY> <PICKY> <PICKY> }

<TYPE>  ::= Number
          | Boolean
          | ( <TYPE> -> <TYPE> )

Γ ⊢ n : Number

Γ ⊢ x : Γ(x)

Γ ⊢ A : Number   Γ ⊢ B : Number
———————————————————————————————
     Γ ⊢ {+ A B} : Number

Γ ⊢ A : Number   Γ ⊢ B : Number
———————————————————————————————
     Γ ⊢ {< A B} : Boolean

          Γ[x:=τ₁] ⊢ E : τ₂
———————————————————————————————————————
Γ ⊢ {fun {x : τ₁} : τ₂ E} : (τ₁ -> τ₂)

Γ ⊢ F : (τ₁ -> τ₂)  Γ ⊢ V : τ₁
——————————————————————————————
     Γ ⊢ {call F V} : τ₂

Γ ⊢ C : Boolean   Γ ⊢ T : τ   Γ ⊢ E : τ
———————————————————————————————————————
          Γ ⊢ {if C T E} : τ
```

Extending Picky

In general, we can extend this language in one of two ways. For example, lets say that we want to add the `with` form. One way to add it is what we did above — simply add it to the language, and write the rule for it. In this case, we get:

$$\frac{\Gamma \vdash V : \tau_1 \qquad \Gamma[x{:=}\tau_1] \vdash E : \tau_2}{\Gamma \vdash \{\texttt{with}\ \{x : \tau_1\ V\}\ E\} : \tau_2}$$

Note how this rule encapsulates information about the scope of `with`. Also note that we need to specify the types for the bound values.

Another way to achieve this extension is if we add `with` as a derived rule. We know that when we see a

```
{with {x V} E}
```

expression, we can just translate it into

```
{call {fun {x} E} V}
```

So we could achieve this extension by using a rewrite rule to translate all `with` expressions into `call`s of anonymous functions (eg, using the `with-stx` facility that we have seen recently). This could be done formally: begin with the `with` form, translate to the `call` form, and finally show the necessary goals to prove its type. The only thing to be aware of is the need to translate the types too, and there is one type that is missing from the typed-with version above — the output type of the function. This is an indication that we don't really need to specify function output types — we can just deduce them from the code, provided that we know the input type to the function.

Indeed, if we do this on a general template for a `with` expression, then we end up with the same goals that need to be proved as in the above rule:

$$\dfrac{\dfrac{\dfrac{\Gamma[x{:=}\tau_1] \vdash E : \tau_2}{\Gamma \vdash \{\texttt{fun}\ \{x : \tau_1\} : \tau_2\ E\} : (\tau_1\ \texttt{->}\ \tau_2)} \qquad \Gamma \vdash V : \tau_1}{\Gamma \vdash \{\texttt{call}\ \{\texttt{fun}\ \{x : \tau_1\} : \tau_2\ E\}\ V\} : \tau_2}}{\Gamma \vdash \{\texttt{with}\ \{x : \tau_1\ V\}\ E\} : \tau_2}$$

Conclusion — we've seen type judgment rules, and using them in proof trees. Note that in these trees there is a clear difference between rules that have no preconditions — there are axioms that are always true (eg, a numeral is always of type `num`).

The general way of proving a type seems similar to evaluation of an expression, but there is a huge difference — *nothing* is really getting evaluated. As an example, we always go into the body of a function expression, which is done to get the function's type, and this is later used anywhere this function is used — when you evaluate this:

```
{with {f {fun {x : Number} : Number x}}
  {+ {call f 1} {call f 2}}}
```

you first create a closure which means that you don't touch the body of the function, and later you use it twice. In contrast, when you prove the type of this expression, you immediately go into the body of the function which you have to do to prove that it has the expected `Number->Number` type, and then you just use this type twice.

Finally, we have seen the importance of using the same type letters to enforce types, and in the case of typing an `if` statement this had a major role: specifying that the two arms can be any two types, or the same type.

Implementing Picky

The following is a simple implementation of the Picky language. It is based on the environments-based Flang implementation. Note the two main functions here — `typecheck` and `typecheck*`.

```
;; The Picky interpreter, verbose version

#lang pl

#|
The grammar:
  <PICKY> ::= <num>
            | <id>
            | { + <PICKY> <PICKY> }
            | { - <PICKY> <PICKY> }
            | { = <PICKY> <PICKY> }
            | { < <PICKY> <PICKY> }
            | { fun { <id> : <TYPE> } : <TYPE> <PICKY> }
            | { call <PICKY> <PICKY> }
            | { with { <id> : <TYPE> <PICKY> } <PICKY> }
            | { if <PICKY> <PICKY> <PICKY> }
  <TYPE>  ::= Num | Number
            | Bool | Boolean
            | { <TYPE> -> <TYPE> }

Evaluation rules:
  eval(N,env)                = N
  eval(x,env)                = lookup(x,env)
  eval({+ E1 E2},env)        = eval(E1,env) + eval(E2,env)
  eval({- E1 E2},env)        = eval(E1,env) - eval(E2,env)
  eval({= E1 E2},env)        = eval(E1,env) = eval(E2,env)
  eval({< E1 E2},env)        = eval(E1,env) < eval(E2,env)
  eval({fun {x} E},env)      = <{fun {x} E}, env>
  eval({call E1 E2},env1)    = eval(B,extend(x,eval(E2,env1),env2))
                              if eval(E1,env1) = <{fun {x} B}, env2>
                             = error!  otherwise <-- never happens
  eval({with {x E1} E2},env) = eval(E2,extend(x,eval(E1,env),env))
  eval({if E1 E2 E3},env)    = eval(E2,env)  if eval(E1,env) is true
                             = eval(E3,env)  otherwise

Type checking rules:

  Γ ⊢ n : Number

  Γ ⊢ x : Γ(x)

  Γ ⊢ A : Number   Γ ⊢ B : Number
  ———————————————————————————————
       Γ ⊢ {+ A B} : Number

  Γ ⊢ A : Number   Γ ⊢ B : Number
  ———————————————————————————————
       Γ ⊢ {< A B} : Boolean

            Γ[x:=τ₁] ⊢ E : τ₂
  ——————————————————————————————————————
  Γ ⊢ {fun {x : τ₁} : τ₂ E} : (τ₁ -> τ₂)

  Γ ⊢ F : (τ₁ -> τ₂)  Γ ⊢ V : τ₁
  ——————————————————————————————
       Γ ⊢ {call F V} : τ₂

  Γ ⊢ V : τ₁   Γ[x:=τ₁] ⊢ E : τ₂
  ——————————————————————————————
   Γ ⊢ {with {x : τ₁ V} E} : τ₂

  Γ ⊢ C : Boolean   Γ ⊢ T : τ   Γ ⊢ E : τ
  ———————————————————————————————————————
            Γ ⊢ {if C T E} : τ

|#

(define-type PICKY
  [Num   Number]
  [Id    Symbol]
  [Add   PICKY PICKY]
  [Sub   PICKY PICKY]
  [Equal PICKY PICKY]
  [Less  PICKY PICKY]
  [Fun   Symbol TYPE PICKY TYPE] ; name, in-type, body, out-type
  [Call  PICKY PICKY]
  [With  Symbol TYPE PICKY PICKY]
  [If    PICKY PICKY PICKY])

(define-type TYPE
  [NumT]
  [BoolT]
  [FunT TYPE TYPE])

(: parse-sexpr : Sexpr -> PICKY)
;; parses s-expressions into PICKYs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)    (Num n)]
    [(symbol: name) (Id name)]
    [(list '+ lhs rhs) (Add   (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '- lhs rhs) (Sub   (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '= lhs rhs) (Equal (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '< lhs rhs) (Less  (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list 'call fun arg)
                       (Call  (parse-sexpr fun) (parse-sexpr arg))]
    [(list 'if c t e)
     (If (parse-sexpr c) (parse-sexpr t) (parse-sexpr e))]
    [(cons 'fun more)
     (match sexpr
       [(list 'fun (list (symbol: name) ': itype) ': otype body)
        (Fun name
             (parse-type-sexpr itype)
             (parse-sexpr body)
             (parse-type-sexpr otype))]
       [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
    [(cons 'with more)
     (match sexpr
       [(list 'with (list (symbol: name) ': type named) body)
        (With name
              (parse-type-sexpr type)
```

```
                (parse-sexpr named)
                (parse-sexpr body))]
         [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
      [else (error 'parse-sexpr "bad expression syntax: ~s" sexpr)]))

(: parse-type-sexpr : Sexpr -> TYPE)
;; parses s-expressions into TYPEs
(define (parse-type-sexpr sexpr)
  (match sexpr
    ['Number  (NumT)]
    ['Boolean (BoolT)]
    ;; allow shorter names too
    ['Num  (NumT)]
    ['Bool (BoolT)]
    [(list itype '-> otype)
     (FunT (parse-type-sexpr itype) (parse-type-sexpr otype))]
    [else (error 'parse-type-sexpr "bad type syntax in ~s" sexpr)]))

(: parse : String -> PICKY)
;; parses a string containing a PICKY expression to a PICKY AST
(define (parse str)
  (parse-sexpr (string->sexpr str)))

;; Typechecker and related types and helpers

;; this is similar to ENV, but it holds type information for the
;; identifiers during typechecking; it is essentially "Γ"
(define-type TYPEENV
  [EmptyTypeEnv]
  [ExtendTypeEnv Symbol TYPE TYPEENV])

(: type-lookup : Symbol TYPEENV -> TYPE)
;; similar to `lookup' for type environments; note that the
;; error is phrased as a typecheck error, since this indicates
;; a failure at the type checking stage
(define (type-lookup name typeenv)
  (cases typeenv
    [(EmptyTypeEnv) (error 'typecheck "no binding for ~s" name)]
    [(ExtendTypeEnv id type rest-env)
     (if (eq? id name) type (type-lookup name rest-env))]))

(: typecheck : PICKY TYPE TYPEENV -> Void)
;; Checks that the given expression has the specified type.
;; Used only for side-effects (to throw a type error), so return
;; a void value.
(define (typecheck expr type type-env)
  (unless (equal? type (typecheck* expr type-env))
    (error 'typecheck "type error for ~s: expecting a ~s"
           expr type)))

(: typecheck* : PICKY TYPEENV -> TYPE)
;; Returns the type of the given expression (which also means that
;; it checks it). This is a helper for the real typechecker that
;; also checks a specific return type.
(define (typecheck* expr type-env)
  (: two-nums : PICKY PICKY -> Void)
  (define (two-nums e1 e2)
    (typecheck e1 (NumT) type-env)
    (typecheck e2 (NumT) type-env))
  (cases expr
    [(Num n) (NumT)]
    [(Id name) (type-lookup name type-env)]
    [(Add   l r) (two-nums l r) (NumT)]
    [(Sub   l r) (two-nums l r) (NumT)]
    [(Equal l r) (two-nums l r) (BoolT)]
    [(Less  l r) (two-nums l r) (BoolT)]
    [(Fun bound-id in-type bound-body out-type)
     (typecheck bound-body out-type
                (ExtendTypeEnv bound-id in-type type-env))
     (FunT in-type out-type)]
    [(Call fun arg)
     (cases (typecheck* fun type-env)
       [(FunT in-type out-type)
        (typecheck arg in-type type-env)
        out-type]
       [else (error 'typecheck "type error for ~s: expecting a fun"
                    expr)])]
    [(With bound-id itype named-expr bound-body)
     (typecheck named-expr itype type-env)
     (typecheck* bound-body
                 (ExtendTypeEnv bound-id itype type-env))]
    [(If cond-expr then-expr else-expr)
     (typecheck cond-expr (BoolT) type-env)
     (let ([type (typecheck* then-expr type-env)])
       (typecheck else-expr type type-env) ; enforce same type
       type)]))

;; Evaluator and related types and helpers

(define-type ENV
  [EmptyEnv]
  [Extend Symbol VAL ENV])

(define-type VAL
  [NumV  Number]
  [BoolV Boolean]
  [FunV  Symbol PICKY ENV])

(: lookup : Symbol ENV -> VAL)
;; lookup a symbol in an environment, return its value or throw an
;; error if it isn't bound
(define (lookup name env)
  (cases env
    [(EmptyEnv) (error 'lookup "no binding for ~s" name)]
    [(Extend id val rest-env)
     (if (eq? id name) val (lookup name rest-env))]))

(: strip-numv : Symbol VAL -> Number)
;; converts a VAL to a Racket number if possible, throws an error if
;; not using the given name for the error message
(define (strip-numv name val)
  (cases val
    [(NumV n) n]
    ;; this error will never be reached, see below for more
    [else (error name "expected a number, got: ~s" val)]))
```

```
(: arith-op : (Number Number -> Number) VAL VAL -> VAL)
;; gets a Racket numeric binary operator, and uses it within a NumV
;; wrapper
(define (arith-op op val1 val2)
  (NumV (op (strip-numv 'arith-op val1)
            (strip-numv 'arith-op val2))))

(: bool-op : (Number Number -> Boolean) VAL VAL -> VAL)
;; gets a Racket numeric binary predicate, and uses it
;; within a BoolV wrapper
(define (bool-op op val1 val2)
  (BoolV (op (strip-numv 'bool-op val1)
             (strip-numv 'bool-op val2))))

(: eval : PICKY ENV -> VAL)
;; evaluates PICKY expressions by reducing them to values
(define (eval expr env)
  (cases expr
    [(Num n) (NumV n)]
    [(Id name) (lookup name env)]
    [(Add   l r) (arith-op + (eval l env) (eval r env))]
    [(Sub   l r) (arith-op - (eval l env) (eval r env))]
    [(Equal l r) (bool-op  = (eval l env) (eval r env))]
    [(Less  l r) (bool-op  < (eval l env) (eval r env))]
    [(Fun bound-id in-type bound-body out-type)
     ;; note that types are not used at runtime,
     ;; so they're not stored in the closure
     (FunV bound-id bound-body env)]
    [(Call fun-expr arg-expr)
     (define fval (eval fun-expr env))
     (cases fval
       [(FunV bound-id bound-body f-env)
        (eval bound-body
              (Extend bound-id (eval arg-expr env) f-env))]
       ;; `cases' requires complete coverage of all variants, but
       ;; this `else' is never used since we typecheck programs
       [else (error 'eval "`call' expects a function, got: ~s"
                    fval)])]
    [(With bound-id type named-expr bound-body)
     (eval bound-body (Extend bound-id (eval named-expr env) env))]
    [(If cond-expr then-expr else-expr)
     (let ([bval (eval cond-expr env)])
       (if (cases bval
             [(BoolV b) b]
             ;; same as above: this case is never reached
             [else (error 'eval "`if' expects a boolean, got: ~s"
                                 bval)])
         (eval then-expr env)
         (eval else-expr env)))]))

(: run : String -> Number)
;; evaluate a PICKY program contained in a string
(define (run str)
  (let ([prog (parse str)])
    (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)]))))

;; tests -- including translations of the FLANG tests
(test (run "5") => 5)
(test (run "{< 1 2}") =error> "type error")
(test (run "{fun {x : Num} : Num {+ x 1}}") =error> "type error")
(test (run "{call {fun {x : Num} : Num {+ x 1}} 4}") => 5)
(test (run "{with {x : Num 3} {+ x 1}}") => 4)
(test (run "{with {identity : {Num -> Num} {fun {x : Num} : Num x}}
              {call identity 1}}")
      => 1)
(test (run "{with {add3 : {Num -> Num}
                   {fun {x : Num} : Num {+ x 3}}}
              {call add3 1}}")
      => 4)
(test (run "{with {add3 : {Num -> Num}
                   {fun {x : Num} : Num {+ x 3}}}
              {with {add1 : {Num -> Num}
                     {fun {x : Num} : Num {+ x 1}}}
                {with {x : Num 3}
                  {call add1 {call add3 x}}}}}")
      => 7)
(test (run "{with {identity : {{Num -> Num} -> {Num -> Num}}
                   {fun {x : {Num -> Num}} : {Num -> Num} x}}
              {with {foo : {Num -> Num}
                     {fun {x : Num} : Num {+ x 1}}}
                {call {call identity foo} 123}}}")
      => 124)
(test (run "{with {x : Num 3}
              {with {f : {Num -> Num}
                     {fun {y : Num} : Num {+ x y}}}
                {with {x : Num 5}
                  {call f 4}}}}")
      => 7)
(test (run "{call {with {x : Num 3}
                    {fun {y : Num} : Num {+ x y}}}
                  4}")
      => 7)
(test (run "{with {f : {Num -> Num}
                   {with {x : Num 3} {fun {y : Num} : Num {+ x y}}}}
              {with {x : Num 100}
                {call f 4}}}")
      => 7)
(test (run "{call {call {fun {x : {Num -> {Num -> Num}}}
                             : {Num -> Num}
                          {call x 1}}
                        {fun {x : Num} : {Num -> Num}
                          {fun {y : Num} : Num {+ x y}}}}
                  123}")
      => 124)
(test (run "{call {fun {x : Num} : Num {if {< x 2} {+ x 5} {+ x 6}}}
                  1}")
      => 6)
(test (run "{call {fun {x : Num} : Num {if {< x 2} {+ x 5} {+ x 6}}}
                         2}")
           => 8)
```

One thing that is very obvious when you look at the examples is that this language is way too verbose to be practical — types are repeated over and over again. If you look carefully at the typechecking fragments for the two relevant expressions — `fun` and `with` — you can see that we can actually get rid of almost all of the type annotations.