Type-checking in the interpreter

Introduction

We will now discuss the implementation of static type-checking.

Let us start by considering what the type-checker might have to do in an example like the following program:

```
(let ([x (+ 3 2)])
  (* x 2))
```

Let us think of what a type-checker may have to do.

- It starts off by looking at a let expression. It will need to start by type-checking the addition.
- When time comes to evaluate the multiplication on the second line, it must have some idea about the type of x. That type should be the result type of the addition operation.
- Of course we could have had an arbitrary expression there instead of an addition.
- What this means is that the typechecker needs to return some type as the result of typechecking an expression: It must both confirm that the expression is "safe" and return what the type of the overall expression must be.
- The other important component is that we need to know what the type of x has to be while typechecking the product on the second line. This is exactly analogous to knowing the value of x when evaluating the body of a let expression. We introduced environments to be able to do exactly that.
- We will therefore need to provide the typechecker with a "static environment", one that relates symbols to types (as opposed to the dynamic environment that related symbols to values/locations).
- Unlike for evaluation, we have no need for a store: A store allows us to change the value at a location. But with a type system in place, if it is to have any meaning, the values stored at a specific location should all have the same type.

So in other words we know what our type-checker needs to do:

- It takes as input a static environment that binds symbols to types, and an expression. We can use the same environment setup from the evaluator, we will just insert types in there instead of values.
- It then checks the expression for type inconsistencies, and throws an error if it finds any.
- If there are no inconsistencies, then the type-checker must determine the type of the overall expression, and it returns that.

Here is the language we will look at. Our functions need to specify the type of their argument. If we want to allow them to recursively call themselves we would need to also specify the return value, or else infer it. We will discuss type inference in the future. For now we will expect functions to specify both argument and return types.

```
(struct varC (s) #:transparent)
(struct numC (n) #:transparent)
(struct boolC (n) #:transparent)
(struct arithC (op e1 e2) #:transparent)
(struct ifC (tst thn els) #:transparent)
(struct pairC (e1 e2) #:transparent)
(struct carC (e) #:transparent)
(struct cdrC (e) #:transparent)
(struct unitC () #:transparent)
(struct letC (s e1 e2) #:transparent)
(struct funC (fname arg ty-arg body ty-body) #:transparent)
(struct callC (e1 e2) #:transparent)
```

We also need to specify similar structures for types. We will need the following types:

- Types for numbers and booleans
- A type for pairs, depending on two types
- A unit type.
- An "arrow" type for functions, depending on two types

```
(struct numT () #:transparent)
(struct boolT () #:transparent)
(struct unitT () #:transparent)
(struct pairT (t1 t2) #:transparent)
(struct funT (t1 t2) #:transparent)     ;; t1 -> t2
```

Here's how the typechecker would start:

```
(define (tc env e)
  (cond
    [(varC? e) (lookup (varC-s) env)]
    [(numC? e) numT]
    [(boolC? e) boolT]
    [(unitC? e) unitT]
    ;; more cases here ... ))
```

We will now consider each of the remaining cases. We will start with the arithmetic operators: They should only apply to numbers, and the result should be a number:

```
    [(arithC? e)
     (if (and (numT? (tc env (arithC-e1 e)))
              (numT? (tc env (arithC-e3 e))))
         numT
         (error "Arithmetic on non-numbers"))]
```

Next we'll handle a conditional. The test component must evaluate to a boolean, and the other two must have the same type. That type is the return type:

```
    [(ifC? e)
     (if (boolT? (tc env (ifC-thn e)))
         (let ([t1 (tc env (ifC-thn e))]
               [t2 (tc env (ifC-els e))])
           (if (equal? t1 t2)
               t1
               (error "Conditional branches must have same type")))
         (error "Conditional test must have boolean type"))]
```

Next we have pairs: They simply form a new type out of their components:

```
[(pairC? e)
 (pairT (tc env (pairC-e1 e))
        (tc env (pairC-e2 e)))]
```

The pair accessors fst and snd are more interesting. They evaluate their expression, and it must be of pair type. They then read off the type from the pair.

```
[(carC? e)
 (let ([t (tc env (carC-e e))])
   (if (pairT? t)
       (pairT-t1 t)
       (error "Trying to call car on non-pair"))))]
[(cdrC? e)
 (let ([t (tc env (cdrC-e e))])
   (if (pairT? t)
       (pairT-t1 t)
       (error "Trying to call cdr on non-pair"))))]
```

Let statements are more interesting. As we discussed at the beginning, we need to typecheck the second expression in an environment that has the symbol bound to the type of the first expression.

```
[(letC? e)
 (tc (bind (letC-s e)
           (tc env (letC-e1 e))
           env)
     (letC-e2 e))]
```

Function definitions turn out to be the hardest case. We will start with function calls: They typecheck their first argument, and it must be a function (arrow) type. The argument type there must agree with the type of the second expression. The whole expression then has the return type stored in the function type.

```
[(callC? e)
 (let ([tf (tc env (callC-e1 e))]
       [tv (tc env (callC-e2 e))])
   (if (funT? tf)
       (if (equal? (funT-t1 tf) tv)
           t2
           (error "Argument type does not match function's expected type"))
       (error "Calling non-function"))))]
```

Finally, we have function definitions. This is an interesting distinction between evaluation and typechecking. For evaluation we had to do the hard work when calling a function, as we had to extend the dynamic environment at that point. The typechecker needs to do its hard work when "creating" the function closure, to ensure that the function definition has the types it claims. This depends only on the type of the argument and on the body expression, and we can check this without needing to call the function.

```
[(funC? e)
 (let ([targ (funC-ty-arg e)]
       [tret (funC-ty-body e)]
       [tbody (tc (funC-body e)
```

```
                (bind (funC-s e) targ env)])
    (if (equal? tret tbody)
        (funT targ tret)
        (error "Suggested return type does not match actual body type"))))]
```