Chapter 14

Types

The first principle was security... A consequence of this principle is that every occurrence of every subscript of every subscripted variable was on every occasion checked at run time against both the upper and the lower declared bounds of the array. Many years later we asked our customers whether they wished us to provide an option to switch off these checks in the interests of efficiency on production runs. Unanimously, they urged us not tothey already knew how frequently subscript errors occur on production runs where failure to detect them could be disastrous. I note with fear and horror that even in 1980, language designers and users have not learned this lesson. In any respectable branch of engineering, failure to observe such elementary precautions would have long been against the law.

Tony Hoare, describing Elliott Brothers' Algol 60 implementation in *The Emperor's Old Clothes*, 1980 Turing Award Speech

The previous chapter illustrated how we can define a language with greater expressiveness by changing one of the evaluation rules. In this chapter, we consider changing our Charme language in a way that does not increase its expressiveness, but in fact reduces it. Reducing expressiveness may seem undesirable, but in fact, much of the effort in modern language design seeks to reduce language expressiveness with the goal of preventing programmers from expressing programs that will crash or produce unexpected results when they are executed. This is especially important when software is used to control a physical device whose correct and continued function is essential to safety. Nuclear power plants, anti-lock brakes, and aircraft avionics are examples of software where it is much better to prevent a programmer from expressing something that may not mean what the programmer intends than to allow maximum expressiveness.

14.1 Types

A type defines a possibly infinite set of values and the operations that can be performed on them. For example, number is a type. We can perform addition, multiplication, and comparisons on values of type number. In Scheme, the primitive procedures are defined to take certain types of values as their inputs. For example, the primitive procedure + takes zero or more numbers as inputs, and the primitive procedure `car` takes a pair (constructed using `cons`) as its one input.

We have used several different types in our Scheme programs so far including numbers (including integer, rational, and floating point numbers), Booleans (`#t` and `#f`), pairs (the result of a `cons`), and strings (sequences of characters enclosed in quotes). Except for Booleans which has only two values, all of the other types have infinitely many possible values (of course, the number of them that could be represented in any finite computer is limited). In addition to the primitive types, we can construct new types. A procedure type is defined by the number and types of the inputs, and the type of the output. We describe procedure types using an arrow (`->`). The input types are listed on the left side of the arrow, and the output type is shown on the right side of the arrow. For example:

- The primitive procedure + has type `(Number Number) -> Number`. It takes two numbers as inputs, and produces a number as output.[1]
- The primitive procedure < has type `(Number Number) -> Boolean`. It takes two numbers as inputs, and produces a Boolean as output.
- The `max` procedure `(lambda (a b) (if (> a b) a b))` has type `(Number Number) -> Number`.

There's no business like show business, but there are several businesses like accounting.
David Letterman

In Scheme (and in Charme), types are *latent*. This means they are not explicitly denoted in the program text. Nevertheless, they are very important for the correct execution of a Scheme program. If the value passed as an operand is not of the correct type, an error will result. For example (in Scheme),

```
> (+ 1 #t)
+: expects type <number> as 2nd argument, given: #t; other
```

[1]In fact, the Scheme and Charme primitive + procedure take zero or more numbers as inputs, so their actual type is `(Number*) -> Number` where the `*` denotes "zero or more".

```
arguments were: 1
> (max 3 (lambda (x) x))
>: expects type <real number> as 2nd argument, given:
     #<procedure>; other arguments were: 3
```

Note that types are check only when an expression is evaluated. For example,

```
> (define (badtype a b) (+ (> a b) b))
> (badtype 3 4)
+: expects type <number> as 1st argument, given: #f; other
arguments were: 4
```

Note that the definition of `badttype` does not produce a type error, even though their are no possible inputs for which it could make sense since the body expression uses the primitive + procedure with a Boolean input. This is *dynamic* type checking. Expressions that are not well typed will produce errors when those expressions are evaluated, but not earlier.

In this chapter, we will modify the Charme language and our interpreter, to change the way types are checked. Instead of using latent types as is done in Scheme, Python, and Charme, we will introduce *manifest* types. This means every definition and parameter will include explicit type information that describe the expected type of the variable or parameter. This requires extra typing and increases the length of the program text, but may make it easier to understand and reason about programs. To provide manifest types, we need to change the language grammar.

In addition, we modify the Charme interpreter to perform *static* type checking. With static type checking, expressions are checked for correct types before they are evaluated. Static type checking reduces the expressiveness of our programming language: it means that some expressions that would have values with dynamic type checking are no longer valid expressions. The advantage of static type checking is that it detects many programming errors early. Whereas dynamic checking checks the types during an execution on *one* path through the program, static checking checks the types before an execution on *all* possible paths through the program. If a correct static type checker deduces the program is well-typed, there is no input on which it could produce a run-time type error. In many cases, it is much better to detect an error early, than to detect it later when a program is running. This is especially true in safety-critical software (such as flight avionics software) where a program failure can have disastrous results (such as a plane crashing).

14.2 Manifest Types

To provide manifest types, we modify the grammar for our language to include type declarations. Two rules need to change: definitions and lambda expressions. We replace the original definition rule,

Definition ::⇒ `(define` ***Name*** *Expression*`)`

with a rule that includes a type specification after the name:

Definition ::⇒ `(define` ***Name*** **:** *Type Expression*`)`

We modify the lambda expression grammar rule to include a type specification for each parameter:

ProcedureExpression ::⇒ `(lambda (` *Parameters* `)` *Expression* `)`
Parameters ::⇒ ϵ | ***Name*** **:** *Type Parameters*

The new nonterminal, *Type*, is used to describe a type specification. A type specification can specify a primitive type. Like a primitive value, a primitive expression is pre-defined by the language and its meaning cannot be broken into smaller parts. StaticCharme supports only two primitive types: `Number` (for representing numbers), and `Boolean` (for representing the Boolean values #t and #f).

Type ::⇒ *PrimitiveType*
PrimitiveType ::⇒ **Number** | **Boolean**

We also need rules for specifying procedure types. The type of a procedure is specified by the type of it inputs (that is, a list of the input types), and the type of its results (in Scheme and Charme, a procedure can only return one value, so the result type is a single type). The arrow (->) symbol is used to denote a procedure type, with the operand type list on the left side of the arrow and the result type on the right side of the arrow. To avoid ambiguity when specifying procedures that have procedure result types, we use parentheses around the procedure type specification.

Type	$::\Rightarrow$	*ProcedureType*
ProcedureType	$::\Rightarrow$	(*ProductType* -> *Type*)
ProductType	$::\Rightarrow$	(*TypeList*)
TypeList	$::\Rightarrow$	*Type TypeList*
TypeList	$::\Rightarrow$	ϵ

The grammar changes mean StaticCharme programs that use definitions or lambda expressions will not be valid Charme programs, and vice-versa. Here are some example definitions in the StaticCharme language:

```
(define x:Number 3)
```

Defines `x` as a variable with type `Number`.

```
(define square:((Number) -> Number)
(lambda (x:Number) (* x x)))
```

Defines `square` as a procedure that takes one input of type `Number`, and produces a `Number` as output.

```
(define compose:((((Number) -> Number)
                  ((Number) -> Number))
                  -> ((Number) -> Number))
  (lambda (f:((Number) -> Number)
           g:((Number) -> Number))
     (lambda (x:Number) (g (f x)))))
```

Defines `compose` as a procedure that takes as inputs two procedures, each of which takes a single `Number` as input and produces a `Number` as output, and produces as output a procedure that takes a single `Number` as input and produces a `Number` as its output.

The last example reveals one of the ways StaticCharme reduces the expressiveness of Charme. In addition to being far longer than the Scheme `compose` procedure, it only works on a small subset of the inputs for which the Scheme `compose` procedure works. It is not possible in StaticCharme to define a single compose procedure that works on as many inputs as the Scheme `compose` procedure, `(lambda (f g) (lambda (x) (g (f x))))`. This procedure takes any two procedures as input, so long as the first one takes a single input, and the second one takes a single input of the same type that is output by the first procedure. Hence, the Scheme `compose` procedure works on inputs of type

```
((Any) -> Boolean) ((Boolean) -> Boolean)
```

such as `(compose zero? not)`. We could define a compose procedure in StaticCharme that takes different types as its inputs, but it would have to be a different procedure from the `compose` procedure defined above.

14.3 Representing Types

To represent types in our StaticCharme interpreter we will use Python classes that correspond to each of the main *Type* grammar rules: *PrimitiveType*, *ProductType*, and *ProcedureType*. To avoid confusion with Python's built-in `type` keyword, we will prepend a `C` (for "Charme") to the names of the corresponding classes.

Figure 14.1 shows the class hierarchy for representing the StaticCharme types. The `CPrimitiveType`, `CProductType`, and `CProcedureType` classes all inherit from the `CType` class.

The `CType` class defines methods for determining if a type is of a given kind (e.g., `isPrimitiveType` returns true only when invoked on a `CPrimitiveType` object. In the superclass, all of these methods are defined to return `False`. The subclass will override the appropriate method to return `True`.

```
class CType:
    # These methods are overridden by subclasses
```

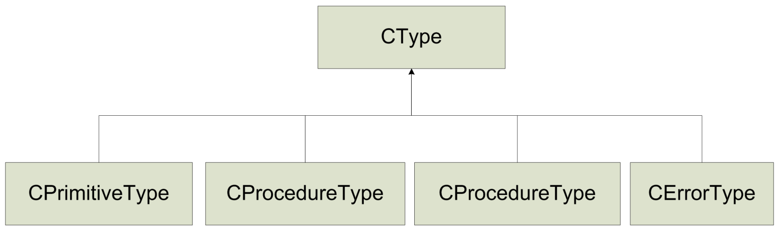

Figure 14.1: Class hierarchy for representing types.

```
    def isPrimitiveType(self):
        return False
    def isProcedureType(self):
        return False
    def isProductType(self):
        return False
    def isError(self):
        return False
```

Primitive Types. The `CPrimitiveType` class is used to represent a primitive type:

```
class CPrimitiveType(CType):
    def __init__(self, s): self._name = s
    def __str__(self): return self._name
    def isPrimitiveType(self): return True
    def matches(self, other):
        return other.isPrimitiveType() \
               and self._name == other._name
```

The class declaration syntax, `class` *`Name`*`(`*`SuperName`*`)` defines *Name* as a subclass of *SuperName*. This means `CPrimitiveType` inherits all the methods defined by the `CType` class. This is similar to how we used `make-subobject` to provide subclassing in Chapter 10.

In StaticCharme there are only two primitive types, `Number` and `Boolean`. To admit the possibility of easily adding more primitive types, however, we represent primitive types using a string that is the type name. The instance variable, `_name` is used to store the name of a primitive type. The primitive number type is constructed using `CPrimitiveType('Number')`.

The `isPrimitiveType` method is defined to always return `True`. This overrides the definition in the `CType` class which always returns `False`.

The `matches` method will be used to determine when two types are compatible. It should return a true value with the parameter passed in as `other` is a type that can be used when this type (the `self` object) is expected. For primitive types, the only type that matches in StaticCharme is the same primitive type. Hence, the `matches` procedure first checks if the `other` object is a primitive type, and if it is, the types match only if they have the same name.

Procedure Types. The `CProcedureType` class represents a procedure type:

```
class CProcedureType(CType):
    def __init__(self, args, rettype):
        self._args = args
        self._rettype = rettype
    def __str__(self):
        return "(" + str(self._args) + \
               " -> " + str(self._rettype) + ")"
    def isProcedureType(self):
        return True
    def getReturnType(self):
        return self._rettype
    def getParameterTypes(self):
        return self._args
    def matches(self, other):
        return other.isProcedureType() \
               and self._args.matches(other._args) \
               and self._rettype.matches(other._rettype)
```

A type matches a procedure type, only if it is also a procedure type, and if both the argument types and the return type match.

Produce Types. The `CProductType` class represents a list of types.

```
class CProductType(CType):
    def __init__(self, types):
        self._types = types
    def __str__(self):
        res = "("
        firstone = True
        for t in self._types:
            if firstone:
                firstone = False
            else:
                res = res + " "
            res = res + str(t)
        res = res + ")"
        return res
    def isProductType(self):
        return True
    def matches(self, other):
        if other.isProductType():
            st = self._types
            ot = other._types
            if len(st) == len(ot):
                for i in range(0, len(st)):
                    if not st[i].matches(ot[i]):
                        return False
                # reached end of loop ==> all matched
                return True
        return False
```

Type Errors. The `CErrorType` class is used to represent type errors. It has an instance variable, `_message`, for describing the type error. Since `CErrorType` represents a type error, it does not match any type and `matches` is defined to always return False.

```
class CErrorType(CType):
    def __init__(self, msg):
        self._msg = msg
    def __str__(self):
        return "<Type Error: " + self._msg + ">"
    def matches(self, other):
        return False
    def isError(self):
        return True
```

Constructing Types. Since we modified the grammar to include type specification, we need some way to take the result of parsing a type specification in the grammar, and turning it into a CType object. To do this we define a procedure parseType that takes as input a list representing the output of parse on a single expression, and returns the corresponding CType object.

```
def parseType(p):
    if isinstance(p, str):
        if p == 'Number':
            return CPrimitiveType('Number')
        elif p == 'Boolean':
            return CPrimitiveType('Boolean')
        else:
            evalError("Undefined type: " + p)
    else:
        if len(p) == 3 and p[1] == '->':
            return CProcedureType(parseType(p[0]), \
                                  parseType(p[2]))
        else:
            types = map(parseType, p)
            return CProductType(types)
```

14.4 Modifying the Evaluator

To support static types, we need to make some modifications to the evaluator. In the next section, we describe the extensions that implement static type checking. In this section, we describe the changes needed to associate types with names. This involves changing the `Environment` and `Procedure` classes to add types, and changing the evaluation rule for definitions and lambda expressions to account for the type specifications. We also need to modify the initial global environment so names in the global environment have associated types.

Environments with types. In StaticCharme, a variable name has not only an associated value but also an associated type. We modify the `Environment` class so that instead of associating just a value with each name in the frame, there is a <type, value> pair.

```
class Environment:
    def __init__(self, parent):
        self._parent = parent
        self._frame = { }
    def addVariable(self, name, typ, value):
        self._frame[name] = (typ, value)
    def _lookupPlace(self, name):
        if self._frame.has_key(name):
            return self._frame[name]
        elif (self._parent):
            return self._parent._lookupPlace(name)
        else:
            return None
    def lookupVariable(self, name):
        return self._lookupPlace(name)[1]
    def lookupVariableType(self, name):
        place = self._lookupPlace(name)
        if place:
            return place[0]
        else:
            return CErrorType("Name not found: " + name)
```

We use a Python tuple (i.e., `(typ, value)`) to represent a pair. Tuple elements are selects identically to list elements: `p[0]` selects the first element and `p[1]` selects the second element in the pair `p`.

The `addVariable` method takes an additional parameter, `typ` that represents the type of the variable.[2] It adds a new entry in the `_frame` that is a tuple of the type and value associated with the name.

The `_lookupPlace` method returns the <type, value> tuple associated with a name. The `lookupVariable` method returns the value part of the place tuple. It is not necessary here to check if the name is defined (although defensive programming practice might suggest performing such a check anyway), since the type checking that will be done before an expression is evaluated ensures that all names used must be defined. The `lookupType` method, on the other hand, needs to deal with the situation where an undefined name is used. It returns a `CErrorType` object that represents and describes the undefined name error.

Definitions. The StaticCharme grammar has a different rule for definitions than standard Scheme because of the added type specification. The evaluation rule needs to be modified accordingly — it should define the variable as before with the addition of its type.

```
def evalDefinition(expr, env):
    name = expr[1]
    typ = parseType(expr[3])
    env.addVariable(name, typ, value)
    value = meval(expr[4], env)
    env.addVariable(name, typ, value)
```

Procedures. The lambda expression in StaticCharme includes a type specification for each parameter name. We modify the `Procedure` class to store this information in a procedure object:

[2]We avoid using the proper spelling, `type`, since that is a keyword in Python.

```
class Procedure:
    def __init__(self, params, typ, body, env):
        self._params = params
        self._body = body
        self._typ = typ
        self._env = env
    def getParamTypes(self):
        return self._typ
    ... # other methods unmodified
```

The evaluation rule for lambda expressions extracts the type specifications from the parameter list and constructs a `Procedure` object:

```
def evalLambda(expr, env):
    params = expr[1]
    paramtypes = []
    paramnames = []
    assert len(params) % 3 == 0
    for i in range(0, len(params) / 3):
        name = params[i*3]
        paramnames.append(name)
        typ = parseType(params[(i*3)+2])
        paramtypes.append(typ)
    return Procedure(paramnames, paramtypes, expr[2], env)
```

Global Environment. The primitives placed in the global environment now must have associated types. We modify the `initializeGlobalEnvironment` procedure to include the appropriate types.

```
def initializeGlobalEnvironment():
    global globalEnvironment
    globalEnvironment = Environment(None)
    globalEnvironment.addVariable('#t', \
       CPrimitiveType('Boolean'), True)
    globalEnvironment.addVariable('#f', \
       CPrimitiveType('Boolean'), False)
    globalEnvironment.addVariable('+', \
       typeFromString('((Number Number) -> Number)'), \
       primitivePlus)
    globalEnvironment.addVariable('-', \
       typeFromString('((Number Number) -> Number)'), \
       primitiveMinus)
    globalEnvironment.addVariable('*', \
       typeFromString('((Number Number) -> Number)'), \
       primitiveTimes)
    globalEnvironment.addVariable('=', \
       typeFromString('((Number Number) -> Boolean)'), \
       primitiveEquals)
    globalEnvironment.addVariable('zero?', \
       typeFromString('((Number) -> Boolean)'), \
       primitiveZero)
    globalEnvironment.addVariable('>', \
       typeFromString('((Number Number) -> Boolean)'), \
       primitiveGreater)
    globalEnvironment.addVariable('<', \
       typeFromString('((Number Number) -> Boolean)'), \
       primitiveLessThan)
```

The `typeFromString` procedure provides a convenient way for specifying a type using a string, as it would be done in a StaticCharme program.

```
def typeFromString(s):
    p = parse(s)
    assert len(p) == 1
    return parseType(p[0])
```

14.5 Checking Types

To implement static type checking, we define a procedure `typecheck` that is analogous to `meval`. Like `meval`, it takes two inputs, an expression and an environment. The result of an application of `typecheck` is the type of the input expression in the input environment. If the expression is not well typed, the result will be a `CErrorType` object. Otherwise, it will be an object that represents the type of the expression. We will modify the `evalLoop` procedure to check the type of every expression before it is evaluated, and only evaluate expressions that are well typed.

The definition of `typecheck` is similar to `meval`. It must deal with all the different kinds of StaticCharme expressions, but instead of evaluating them it checks their type.

```
def typecheck(expr, env):
    if isPrimitive(expr):
        return typePrimitive(expr)
    elif isConditional(expr):
        return typeConditional(expr, env)
    elif isLambda(expr):
        return typeLambda(expr, env)
    elif isDefinition(expr):
        return typeDefinition(expr, env)
    elif isName(expr):
        return typeName(expr, env)
    elif isApplication(expr):
        return typeApplication(expr, env)
    else:
        evalError ("Unknown expression type: " + str(expr))
```

To implement type checking, we need to implement the appropriate `type`*`Element`* procedure. We describe all of the procedures next, except for `typeConditional`, which is left as Exercise 14.5.

Primitives. The `typePrimitive` procedure takes a primitive expression and outputs its type.

```
def typePrimitive(expr):
    if isNumber(expr):
        return CPrimitiveType('Number')
    elif isinstance(expr, bool):
     return CPrimitiveType('Boolean')
    elif callable(expr):
        return globalEnvironment.reverseLookupType(expr)
    else:
        assert False
```

The type of number and Boolean expressions is straightforward. Dealing with primitive procedures is more difficult. They are represented by Python procedures, so the predicate `callable(expr)` is true if `expr` is a primitive procedure. But, there is no easy way to determine the type of the procedure.

The solution we use is to search the global environment for a name whose value matches the primitive procedure, and use the associated type. This is done by adding a `reverseLookupType` method to the `Environment` class that finds the type of the place whose value matches a given value. Since all primitive procedures are associated with names in the global environment, we can use the reverse lookup to find the type of a primitive procedure.

```
class Environment:
    ... # other methods
    def reverseLookupType(self, val):
        for entry in self._frame.itervalues():
            if entry[1] == val: return entry[0]
        return CErrorType("Value not found")
```

Names. The type of a name is stored in the environment:

```
def typeName(expr, env):
    return env.lookupVariableType(expr)
```

Definitions. A definition has no type (since it has no value), but type checking a definition may still detect a type error. Hence, `typeDefinition` either returns `None` (to represent no type) or a `CErrorType` object if the definition is mistyped. The type checking rule for a definition is that the type of the value expression must match the specified type.

One subtlety is caused by our desire to support recursive definitions. The value subexpression may use the name that is being defined. This means that we should type check the value subexpression in an environment in which the new name is defined with the specified type. We do this by creating a new environment, whose parent is the input environment, and adding the variable to the environment with value `None`. The value of the name is not yet known, but type checking does not depend on its value, so any value will do here. We create a new environment instead of using the input environment since if the definition is not well-typed, it should not be added to the environment. Names are added to the environment only when the definition is evaluated.

```
def typeDefinition(expr, env):
    assert isDefinition(expr)
    if len(expr) != 5:
        evalError ("Bad definition: %s" % str(expr))
    name = expr[1]
    if isinstance(name, str):
        if expr[2] != ':':
            return \
              CErrorType ("Definition missing type: %s" \
                          % str(expr))
        typ = parseType(expr[3])
        newenv = Environment(env)
        newenv.addVariable(name, typ, None)
        etyp = typecheck(expr[4], newenv)
        if not typ.matches(etyp):
            return CErrorType \
            ("Mistyped definition: %s " \
             + "declared type %s, actual type %s" \
             % (name, typ, etyp))
        return None # definition has no type
    else:
        return CErrorType ("Bad definition: %s" % str(expr))
```

To find the type of the value subexpression (`etyp`), the procedure recursively calls `typecheck` on the subexpression. If the result does not match the specified type (`typ`), the definition is mistyped and a `CErrorType` object is returned. Note that this outcome also results if the value subexpression itself is not well-typed, since in that case the call to `typecheck(expr[4], env)` will result in a `CErrorType` object which does not match any type.

Applications. An application expression is well-typed only if the first subexpression is a procedure. To be well-typed, the operand expressions must match the parameter types for that procedure.

```
def typeApplication(expr, env):
    proctype = typecheck(expr[0], env)
    if not proctype.isProcedureType():
        return \
           CErrorType("Application of non-procedure: " \
                      + str(expr[0]))
    optypes = map (lambda op: typecheck(op, env), expr[1:])
    optype = CProductType(optypes)
    if not proctype.getParameterTypes().matches(optype):
        return \
           CErrorType("Parameter type mismatch: " \
                      + "expected %s, given %s" \
                 % (proctype.getParameterTypes(), optype))
    return proctype.getReturnType()
```

The `optypes` variable is initialized to the result of mapping the `typecheck` procedure on each of the operand subexpressions. This produces a list of type objects (possibly including `CErrorType` objects if any of the operand subexpressions is not well-typed). A `CProductType` object is constructed from the operand type list. The if statement checks if the operand types match the expected parameter types for the procedure. If they match, the type of the application expression is the return type of the applied procedure.

Lambda Expressions. To type check a lambda expression, we need to determine the type that will result from an application of the procedure. This involves partially applying the procedure even though we have no operands. Instead, we create a new environment containing the places corresponding to each parameter. Because of the type specifications, their types are known, but no value is available since the procedure is not being applied. We use `None` for the value, but rely on the nature of type checking. Type checking should never depend on actual values, only on the types which are known because of the parameter type specifications.

If you steal property, you must report its fair market value in your income in the year you steal it unless in the same year, you return it to its rightful owner.
Your Federal Income Tax, IRS Publication 17, p. 90.

```
def typeLambda(expr, env):
    assert isLambda(expr)
    if len(expr) != 3:
        evalError ("Bad lambda expression: %s" % str(expr))
    newenv = Environment(env)
    params = expr[1]
    paramnames = []
    paramtypes = []
    assert len(params) % 3 == 0
    for i in range(0, len(params) / 3):
        name = params[i*3]
        assert params[(i*3)+1] == ':'
        typ = parseType(params[(i*3)+2])
        paramnames.append(name)
        paramtypes.append(typ)
        newenv.addVariable(name, typ, None)
    resulttype = typecheck(expr[2], newenv)
    return CProcedureType(CProductType(paramtypes), \
                          resulttype)
```

The type of the lambda expression is a procedure type with the parameter types given in the parameter type specifications, and the result type determined by type checking the partial application.

Evaluator Loop. The `evalLoop` procedure is changed to type check every expression before it is evaluated. If the expression is not well-typed, type checking produces an error which is displayed. Otherwise, the expression is then evaluated.

The inner loop of the evaluated is replaced with:

```
for expr in exprs:
   typ = typecheck(expr, globalEnvironment)
   if typ and typ.isError():
       print "Error: " + typ.getMessage()
   else:
       res = meval(expr, globalEnvironment)
       if res != None:
           print str(res)
```

Exercise 14.1. Define the `typeConditional(expr, env)` procedure that checks the type of a conditional expression. It should check that all of the predicate expressions evaluate to a Boolean value. In order for a conditional expression to be type correct, the consequent expressions of each clause produce values of the same type. The type of a conditional expression is the type of all of the consequent expressions. ♢

Exercise 14.2.(⋆⋆) A stronger type checker would require that at least one of the conditional predicates must evaluate to a true value. Otherwise, the conditional expression does not have the required type (instead, it produces a run-time error). Either modify your `typeConditional` procedure to implement this stronger typing rule, or explain why it is impossible to do so. ♢

14.5.1 Examples

Here are some example evolutions using our StaticCharme interpreter:

```
    StaticCharme> (define b:Boolean 3)
```

```
Error: Mistyped definition: b declared type Boolean,
    actual type Number
StaticCharme> (define f:((Number) -> Number)
                   (lambda (x: Number) (> x 3)))
Error: Mistyped definition: f declared type ((Number) -> Number),
    actual type ((Number) -> Boolean)
StaticCharme> (define sapp:(() -> Number)
                   (lambda ()
                       (square #t)))
Error: Mistyped definition: sapp declared type (() -> Number),
    actual type (() -> <Type Error: Parameter type mismatch:
    expected (Number), given (Boolean)>)
```

Note that errors are reported for definitions, not when the defined names are used.

Exercise 14.3. Give the type of each of the following StaticCharme expressions. If the expression is not well-typed, explain the type error. Assume the `square` and `compose` definitions from Section 14.2.

a. `(+ 1 #t)`

b. `compose`

c. `(compose square square)`

d. `(compose (lambda (a:Boolean) a) square)`

e. `(+ 1 2 3)`

f. `(square ((lambda (a:Number b:Number) (> a b)) 3 4))`

g.
```
(define infinite-loop:(() -> Number)
        (lambda () (infinite-loop)))
```

h. `(square (infinite-loop))`

◇

14.6 Summary

Modifying our Charme interpreter to implement a language with manifest types and static type checking reduces the expressiveness of the language in ways that make it possible to detect programming errors earlier and more reliably. For software on which people depend, it is far better to detect an error before a program is executed, than to encounter an error at run-time, or to produce an incorrect result.