14
Type Checking

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 to Ů– they 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, *The Emperor's Old Clothes*, 1980 Turing Award Speech

In this chapter, we change the Charme language in a way that does not increase its expressiveness, but in fact *reduces* it. Reducing expressiveness may seem undesirable, but in fact, a great deal of effort in modern language design seeks to reduce language expressiveness.

The reason language designers want to reduce expressiveness it can contribute to the goal of preventing programmers from expressing programs that will crash or produce unexpected results when they are executed. Such languages sacrifice expressiveness for the sake of *truthiness*[1]—increasing the likelihood that a program means what its programmer intends.

A high level of truthiness is important when software is used to control a physical device whose correct and continued function is essential to safety such as software controlling a nuclear power plant, anti-lock brakes, or aircraft avionics. In such cases, it is much better to reduce the expressiveness of your programming language and get more errors when the software is developed, than to get unexpected behaviors when the software is running.

[1]According to Merriam-Webster's dictionary, *truthiness* has two different meanings: (1) "truth that comes from the gut, not books" (due to Stephen Colbert); and (2) "the quality of preferring concepts or facts one wishes to be true, rather than concepts or facts known to be true" (due to the American Dialect Society). Neither of these definitions quite matches how we use the word here to mean the actual meaning of the program is truthful to the programmer's intended meaning, but I know of no English word that better matches the intended meaning.

14.1 Types of Typing

As introduced in Section 5.1, a type defines a possibly infinite set of values and the operations that can be performed on them. For example, Number is a type. Addition, multiplication, and comparisons can be performed on Numbers.

latent

In Scheme (and in Charme and LazyCharme), types are *latent*: 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,

```
> (+ 1 true)
+: expects type <number> as 2nd argument, given: true;
other arguments were: 1
```

Scheme has fairly strong type checking. If the types of operands do not match the expected types, it reports an error. Other languages, such as Python, have weaker type checking:

```
≫ 1 + True
2
```

Instead of reporting a type error when + is used with a Boolean value, Python interprets the value of True as 1 and produces a result.

dynamic type checking

Both Scheme and Python have *dynamic type checking*. This means types are checked only when an expression is evaluated. For example, evaluating (**define** (*badtype a b*) (+ (> *a b*) *b*)) 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. Expressions that are not well-typed produce errors when those expressions are evaluated, but not earlier.

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

In this chapter, we develop an interpreter for the language *StaticCharme* that manages types differently from the Charme language. Instead of using latent types as is done in Scheme, Python, and Charme, StaticCharme uses *manifest* types: every definition and parameter will include explicit type information that describes the expected type of the variable or parameter. This increases the length of the program text and sacrifices expressiveness, but makes it easier to understand and reason about programs. To provide manifest types, we change the language grammar to support explicit type declarations. Many popular programming languages have manifest types including Ada, C, C#, and Java.

manifest

We also modify the Charme interpreter to perform *static type checking*. Static type checking checks that expressions are well-typed when they are defined, rather than waiting until they are evaluated. Static type checking reduces the expressiveness of our programming language; some expressions that would have values with dynamic type checking are no longer valid expressions.

static type checking

Ariane 5

Rocket after run-time error

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 before any path is executed. 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,

$$\textit{Definition} ::\Rightarrow (\textbf{define}\ \textit{Name Expression})$$

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

$$\textit{Definition} ::\Rightarrow (\textbf{define}\ \textit{Name}\ \textbf{:}\ \textit{Type Expression})$$

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

$$\begin{array}{lll} \textit{Expression} & ::\Rightarrow & \textit{ProcedureExpression} \\ \textit{ProcedureExpression} & ::\Rightarrow & (\textbf{lambda}\ (\textit{Parameters})\ \textit{Expression}) \\ \textit{Parameters} & ::\Rightarrow & \epsilon \mid \textit{Name}\ \textbf{:}\ \textit{Type Parameters} \end{array}$$

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 *true* and *false*).

$$\begin{array}{lll} \textit{Type} & ::\Rightarrow & \textit{PrimitiveType} \\ \textit{PrimitiveType} & ::\Rightarrow & \textbf{Number} \mid \textbf{Boolean} \end{array}$$

We also need constructs 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 (a Charme procedure can only return one value, so the result type is a single type). The arrow ($\rightarrow$, or -> in program text) 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.

$$\begin{array}{lll} \textit{Type} & ::\Rightarrow & \textit{ProcedureType} \\ \textit{ProcedureType} & ::\Rightarrow & \textbf{(}\ \textit{ProductType} \rightarrow \textit{Type}\ \textbf{)} \\ \textit{ProductType} & ::\Rightarrow & \textbf{(}\ \textit{TypeList}\ \textbf{)} \\ \textit{TypeList} & ::\Rightarrow & \textit{Type TypeList} \\ \textit{TypeList} & ::\Rightarrow & \epsilon \end{array}$$

For example,

(**define** *x* **: Number** 3)

defines *x* as a Number and initializes its value to 3.

The definition,

```
(define square : ((Number) → Number)
  (lambda (x : Number) (∗ x x)))
```

defines *square* as a procedure that takes one Number input and produces a Number as its output.

Here is a definition of a *compose* procedure that takes two procedures as inputs and produces a procedure as its output:

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

The *compose* example reveals one of the ways StaticCharme reduces the expressiveness of Charme. In addition to requiring more than twice as many characters to define as the Scheme *compose* procedure, this procedure only works on a small subset of the inputs for which the Scheme *compose* procedure works, namely procedures that take Number inputs and produce a Number output.

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*)))). For example, the Scheme *compose* procedure works on inputs of type (((*Any*) → **Boolean**) ((**Boolean**) → **Boolean**)) such as (*compose zero? not*). In StaticCharme, we would need do define different composition procedures for each input type.

14.3 Representing Types

We use Python classes to represent types in our StaticCharme interpreter. 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 the Python classes used to represent StaticCharme types. The *CPrimitiveType*, *CProductType*, *CProcedureType*, and *CErrorType* classes all inherit from the *CType* class.

The *CType* class defines methods for determining if a type is of a given kind

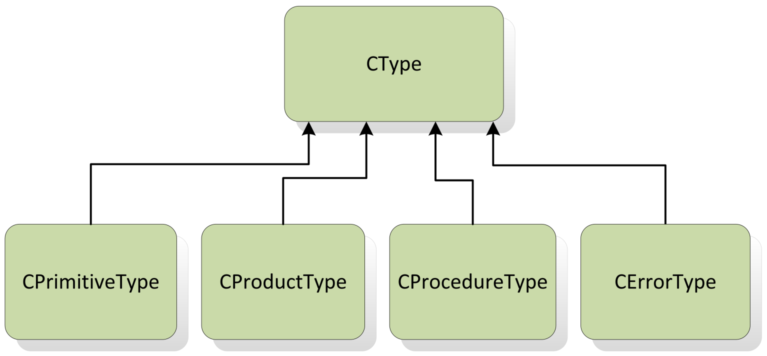

Figure 14.1. Class hierarchy for representing types.

(e.g., *isPrimitiveType* returns true only when invoked on a *CPrimitiveType* object. In the superclass, all of these methods are defined to return False.

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

The subclass will override the appropriate method to return True, and define additional methods to implement the type. The most interesting method each subclass provides is the *matches* method. It takes a *CType* object as its input, as well as the *self* object, and outputs True if the input type matches the self type.

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** *CPrimitiveType*(*CType*), defines the class *CPrimitiveType* as a subclass of *CType*; *CPrimitiveType* inherits all the methods defined by the *CType* class. This is similar to how we used *make-subobject* to provide subclassing in Chapter 11.

In StaticCharme there are only two primitive types, **Number** and **Boolean**. To support easily adding more primitive types, however, we represent primitive types using a string that is the type name. The instance variable, *_name* stores the name of a primitive type. The constructor method takes a string that names the primitive type. So, the primitive number type is constructed using *CPrimitiveType*('Number').

The *__str__* method outputs a string representation of the object, in this case, just its name. The Python syntax *str*(*obj*) invokes the *__str__* method of *obj*. This is useful for printing objects in a human-readable way.

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

The *matches* method is used to determine when two types are compatible. It returns True only if the *other* type is the same primitive type by first checking if the *other* object is a primitive type, and if it is, checking if they have the same name. The Python **and** expression is similar to the Scheme special form: the second expression is only evaluated if the first expression is true.

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)
```

For procedure types to match, both the argument types and the return type must match.

Product 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): # number of types must match
                for i in range(0, len(st)):
                    if not st[i].matches(ot[i]): return False
                # reached end of loop; all matched so types match
                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 always returns False.

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

Constructing Types. Since we modified the grammar to include type specifications, we also need a procedure that takes the result of parsing a type specification and produces 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: # must be product type
      return CProductType(map(parseType, p))
```

14.4 Modifying the Evaluator

This section describe the changes to the interpreter made to associate types with names. We change the *Environment* and *Procedure* classes to add types, and modify the evaluation rule for definitions and lambda expressions to account for the type specifications. We also need to set up the initial global environment so names in the global environment have associated types. Section 14.5 describes the extensions to implement static type checking.

Typed environments. In StaticCharme, a variable name has both an associated value and an associated type. We modify the *Environment* class so that instead of associating just a value with each name in the frame, each place holds a <*type*, *value*> pair.

```
class Environment:
  def __init__(self, parent):
    self._parent = parent
```

```
        self._frame = { }
    def addVariable(self, name, typ, value): # added typ parameter
        self._frame[name] = (typ, value) # replaced value with tuple
    def _lookupPlace(self, name): # added
        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): # rewritten to use _lookupPlace
        return self._lookupPlace(name)[1]
    def lookupVariableType(self, name): # added
        place = self._lookupPlace(name)
        if place: return place[0]
        else: return CErrorType("Name not found: " + name)
```

We use a Python tuple, (*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.[3] The *lookupVariable* method returns the value associated with a variable.[4] The *lookupVariableType* 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 is modified accordingly to define the variable with the addition of its type.

```
def evalDefinition(expr, env):
    assert isDefinition(expr)
    assert len(expr) == 4 and expr[2] == ':'
    env.addVariable(expr[1], parseType(expr[3]), meval(expr[4], env))
```

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

[3]The _ at the beginning of the method name hides the method from external users of the class. Methods beginning with a _ can only be called from within the class implementation. This supports data abstraction; by hiding the *_lookupPlace* method we avoid exposing the tuple representation of (*typ*, *value*) pairs.

[4]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.

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:

```
class Procedure:
  def __init__(self, params, typ, body, env):
    self._params = params
    self._body = body
    self._typ = typ # added
    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):
  assert isLambda(expr)
  assert len(expr) == 3
  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)
```

We also need to change the *mapply* procedure to put the types of the parameters in the new frame.

```
def mapply(proc, operands):
  if (isPrimitiveProcedure(proc)):
    return proc(operands)
  elif isinstance(proc, Procedure):
    params = proc.getParams()
    types = proc.getParamTypes() # added
    newenv = Environment(proc.getEnvironment())
    if len(params) != len(operands):
      evalError ('Parameter length mismatch: %s given operands %s'
             % (str(proc), str(operands)))
    for i in range(0, len(params)):
      newenv.addVariable(params[i], types[i], operands[i]) # added type
```

```
        return meval(proc.getBody(), newenv)
    else:
        evalError('Application of non−procedure: %s' % (proc))
```

Global Environment. The primitives placed in the global environment now must have associated types, so *initializeGlobalEnvironment* is modified to include the appropriate types.

```
def initializeGlobalEnvironment():
    global globalEnvironment
    globalEnvironment = Environment(None)
    globalEnvironment.addVariable('true', CPrimitiveType('Boolean'), True)
    globalEnvironment.addVariable('false', 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 \
      ('<', 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 is an *CErrorType* object. Otherwise, it will be an object that represents the type of the expression.

The definition of *typecheck* is similar to *meval*. It has a clause for each kind of StaticCharme expression, but instead of evaluating the expression by calling *evalExpr* it checks its type using *typeExpr*.

```
def typecheck(expr, env):
    if isPrimitive(expr): return typePrimitive(expr)
    elif isIf(expr): return typeIf(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 implement the *typeExpr* procedure for each language construct. We describe primitives and if expressions first, and then describe the procedures for typechecking names, definitions, applications, and lambda expressions.

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")
```

If. The *typeIf* procedure checks an if-expression is well-typed and outputs the type of its value. The predicate must have type Boolean. The consequent and alternate expressions may have any type, but their types must be the same. The type of the if-expression is the type of consequence and alternative expressions.

```
def typeIf(expr, env):
  if len(expr) != 4: evalError('Badly formed if: ' + str(expr))

  # Predicate must be a Boolean
  predtype = typecheck(expr[1], env)
  if not CPrimitiveType('Boolean').matches(predtype):
    return CErrorType('Mistyped predicate: ' + str(predtype))

  # Types of consequent and alternate expressions must match
  constype = typecheck(expr[2], env)
  alttype = typecheck(expr[3], env)
  if not constype.matches(alttype):
    return CErrorType('Inconsistent branch types: %s, %s'
                % (constype, alttype))
  return constype
```

Names. The type of a name is stored in the environment, so we can type check a name by looking up its type in the environment.

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

The *Environment.lookupVariableType* method produces an evaluation error if the name is not defined, so *typeName* also checks that all names are defined.

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 support this by creating a new environment whose parent is the input environment that contains the variable with its declared type and 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) # Support recursive definitions
      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.

```
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):
    assert params[(i*3)+1] == ':'
    name = params[i*3]
    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.

```
def evalLoop():
  initializeGlobalEnvironment()
  while True:
    inv = raw_input('StaticCharme> ')
    if inv == 'quit': break
```

```
for expr in parse(inv):
  typ = typecheck(expr, globalEnvironment) # added
  if typ and typ.isError(): # added
     print 'Error: ' + typ.getMessage()
  else:
     res = meval(expr, globalEnvironment)
     if res != None: print str(res)
```

Examples. Here are some example evaluations using our StaticCharme interpreter:

```
StaticCharme> (define n : Number 3)
StaticCharme> (define b : Boolean 3)
Error: Mistyped definition: b declared type Boolean, actual type Number
StaticCharme> (define square : ((Number) -> Number)
                     (lambda (x : Number) (* x x)))
StaticCharme> (square 3)
9
StaticCharme> (square true)
Error: Parameter type mismatch: expected (Number), given (Boolean)
StaticCharme> (square 3 5)
Error: Parameter type mismatch: expected (Number), given (Number 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 true)))
Error: Mistyped definition: sapp declared type (() -> Number), actual type (()
     -> <Type Error: Parameter type mismatch: expected (Number), given
     (Boolean)>)
StaticCharme> (if (< 3 4) true false)
True
StaticCharme> (if (+ 3 4) true 6)
Error: Mistyped predicate: Number
StaticCharme> (if (< 3 4) true 6)
Error: Inconsistent branch types: Boolean, Number
StaticCharme> (define fact:((Number) -> Number)
                     (lambda (n:Number) (if (= n 1) 1 (* n (fact (- n 1))))))
StaticCharme> (fact 10)
3628800
```

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

Exercise 14.1. 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 *true*)

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*))

Exercise 14.2. [⋆] Extend the interpreter to support a conditional special form similar to the Scheme **cond** expression. The *typeConditional* procedure 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 must be of the same type. The type of a conditional expression is the type of each consequent expression.

Exercise 14.3. [⋆] 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. (Hint: see Chapter 15.)

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 safety-critical software, 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.